

**AFRL-RI-RS-TR-2009-112**  
**Final Technical Report**  
**April 2009**



# **SCALABLE KNOWLEDGE DISCOVERY THROUGH GRID WORKFLOWS**

University of Southern California

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

STINFO COPY

**AIR FORCE RESEARCH LABORATORY  
INFORMATION DIRECTORATE  
ROME RESEARCH SITE  
ROME, NEW YORK**

## NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88<sup>th</sup> ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2009-112 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/

/s/

DEBORAH A. CERINO  
Work Unit Manager

JOSEPH CAMERA, Chief  
Information & Intelligence Exploitation Division  
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

**REPORT DOCUMENTATION PAGE***Form Approved*  
**OMB No. 0704-0188**

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

**PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.****1. REPORT DATE (DD-MM-YYYY)**

APR 09

**2. REPORT TYPE**

Final

**3. DATES COVERED (From - To)**

Sep 06 – Mar 09

**4. TITLE AND SUBTITLE**SCALABLE KNOWLEDGE DISCOVERY THROUGH GRID  
WORKFLOWS**5a. CONTRACT NUMBER**

FA8750-06-C-0210

**5b. GRANT NUMBER****5c. PROGRAM ELEMENT NUMBER**

31011G

**6. AUTHOR(S)**Yolanda Gil, Ewa Deelman, Jihie Kim, Paul Groth, Gaurang Mehta, Varun  
Ratnakar and Karan Vahi**5d. PROJECT NUMBER**

TNGR

**5e. TASK NUMBER**

00

**5f. WORK UNIT NUMBER**

06

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**University of Southern California  
Information Sciences Institute  
4676 Admiralty Way  
Marina del Rey CA 90292**8. PERFORMING ORGANIZATION  
REPORT NUMBER****9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**AFRL/RIED  
525 Brooks Rd.  
Rome NY 13441-4505**10. SPONSOR/MONITOR'S ACRONYM(S)****11. SPONSORING/MONITORING  
AGENCY REPORT NUMBER**  
AFRL-RI-RS-TR-2009-112**12. DISTRIBUTION AVAILABILITY STATEMENT**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW 2009-1531

**13. SUPPLEMENTARY NOTES****14. ABSTRACT**

The goal of this effort was to drastically reduce the human effort required to configure and execute new workflows for data analysis from weeks to minutes by eliminating the need for costly human monitoring and intervention. This involves developing end-to-end data analysis systems to analyze data from many different sources and with many different algorithms and analytical tools. Their approach combined three central ideas: 1) workflows with rich representations of algorithmic requirements and data products, 2) semantic representations to enable automatic generation of complex workflows, and 3) grid computing to manage the high-performance of many workflows in distributed cross-organization environments.

**15. SUBJECT TERMS**

Computational workflows, grid computing, semantic representations

**16. SECURITY CLASSIFICATION OF:****a. REPORT**  
U**b. ABSTRACT**  
U**c. THIS PAGE**  
U**17. LIMITATION OF  
ABSTRACT**

UU

**18. NUMBER  
OF PAGES**

102

**19a. NAME OF RESPONSIBLE PERSON**

Deborah Cerino

**19b. TELEPHONE NUMBER (Include area code)**

N/A

# TABLE OF CONTENTS

|  |    |
|--|----|
| 1. Goals of System Research in the Tangram Program .....                                 | 1  |
| 2. Architecture of the Workflow System .....   | 2  |
| 2.1 Major Components of the Workflow System .....  | 3  |
| 2.2 The Wings/Pegasus Workflow System.....   | 4  |
| 2.3 Middleware Services Used by the Workflow System .....                                | 6  |
| 2.4 Workflow Execution Facilities .....  | 7  |
| 3. Research Results .....  | 8  |
| 3.1 Automatic Workflow Generation .....  | 8  |
| 3.1.1 Data, Components, Workflows, and Workflow Requests: Requirements and Examples..... | 9  |
| 3.1.2 Formalization .....  | 17 |
| 3.1.3 Algorithm: Automatic Template-Based Workflow Generation .....                      | 29 |
| 3.1.4 A walkthrough Example of Workflow Generation.....                                  | 40 |
| 3.2 Workflow Ranking and Selection.....  | 49 |
| 3.3 Managing Creation and Execution of Multiple Workflows .....                          | 49 |
| 4. Support for Program Integration.....  | 51 |
| 4.1. SR-12 Workflow Generation API.....  | 52 |
| 5. Major Milestones: Demonstrations and Evaluations .....                                | 56 |
| 5.1 SR-6 Demonstration.....  | 56 |
| 5.2 SR-12 Demonstration.....   | 63 |
| 5.2.1 SR-12 Functional Requirement Relevant to SR .....                                  | 64 |
| 5.2.2 SR-12 Technical Requirements Relevant to SR .....                                  | 66 |
| 5.3 SE-18 Evaluation .....   | 68 |
| 5.3.1 SE-18 Metrics and Requirement Relevant to SR.....                                  | 68 |
| 6. Support for Program Evaluations.....  | 71 |
| 6.1 Program Testbed .....  | 71 |
| 6.2 Logging Support .....  | 73 |
| 6.2.1 Logging Format .....   | 73 |
| 6.2.2 Logging Ontology.....  | 74 |
| 6.2.3 Extension to the Workflow Generation API for Logging.....                          | 75 |
| 6.2.4 Client Side Logging Library .....  | 75 |
| 7. Software Releases .....   | 76 |
| 7.1 Installation of the SR Workflow System .....   | 76 |
| 7.1.1. Required Software .....   | 76 |
| 7.1.2. Build and Install Wings .....   | 76 |
| 7.1.3. Build Pegasus.....  | 78 |
| 7.1.4. Install Pegasus.....  | 78 |
| 7.1.5. Build Ensemble Manager.....   | 79 |
| 7.1.6. Install Ensemble Manager.....   | 80 |
| 7.1.7. Create the Ensemble Data Base (DB).....   | 80 |
| 7.1.8. Edit the Ensemble configuration file .....  | 81 |
| 7.1.9. Edit Log4j.configuration file .....   | 81 |
| 7.2 Grid Services.....   | 82 |

|   |    |
|---|----|
| 7.2.1 Linux Headnode Installation.....                      | 82 |
| 7.2.2 Linux Cluster Node Installation.....                  | 89 |
| 7.2.3 Windows Node Installation.....                        | 89 |
| 8. Interim Project Reports and Documentation Released ..... | 89 |
| 9. Conclusions.....   | 92 |
| 10. References.....   | 93 |
| 11. Acronyms.....   | 96 |

## LIST OF FIGURES

|  |    |
|--|----|
| Figure 1: SR Architecture .....  | 3  |
| Figure 2: Middleware Services for the Workflow System .....                              | 7  |
| Figure 3: Stages During Workflow Generation .....  | 10 |
| Figure 4: The representation of a Workflow to learn a model and then classify data. .... | 13 |
| Figure 5: A workflow to learn a model to predict weather .....                           | 14 |
| Figure 6: Structure of the DAG constructed by the Ensemble manager .....                 | 50 |
| Figure 7: Workflows Demonstrated in December 2006 .....                                  | 60 |
| Figure 8: Workflow Creation in SR-6 from Template to Instance to Execution .....         | 61 |
| Figure 9: Example Workflow with Parallel Seed Creation .....                             | 62 |
| Figure 10 : Supporting Data Reuse Across Workflows .....                                 | 63 |
| Figure 11: Tangram Testbed .....   | 72 |

## LIST OF TABLES

|   |    |
|---|----|
| Table 1: Representation of Abstract and Concrete Workflow Components .....  | 11 |
| Table 2: Formal Representation of a Request .....   | 28 |
| Table 3: Top-Level Algorithm for Automatic Template-Based Workflow Generation..   | 31 |
| Table 4: Algorithm for Seeding a Workflow Template with the Seed Given in the<br>Request .....  | 32 |
| Table 5: Algorithm for Backward Sweep Through Workflow .....  | 33 |
| Table 6: Algorithm for Backward Sweep through Components .....  | 34 |
| Table 7: Algorithm for Binding Workflows by Selecting Input Data .....  | 35 |
| Table 8: Algorithm for Forward Sweep Through Workflow .....   | 35 |
| Table 9: Algorithm for Forward Sweep Through Components .....   | 37 |
| Table 10: Algorithm for Estimating Workflow Performance Through the Workflow....  | 38 |
| Table 11: Algorithm for Estimating Workflow Performance Through Components .....  | 38 |
| Table 12: Algorithm for Instantiating Workflows .....   | 39 |
| Table 13: Summary of functions that need to be supported in the metadata services and<br>the component services to enable automatic workflow generation ..... | 41 |
| Table 14: A Workflow Example in N3 Notation .....   | 43 |
| Table 15: Example of Workflow Requests in N3 Notation .....   | 44 |
| Table 16: Relevant Excerpts of a Seeded Workflow .....  | 44 |
| Table 17: An Example Binding-Ready Workflow After the Backward Sweep .....  | 45 |
| Table 18: Relevant Excerpts of an Example Bound Specialized Workflow .....  | 46 |
| Table 19: A Workflow Instance After Workflow Instantiation .....  | 47 |
| Table 20: Example Ground Workflow Generated from a Workflow Instance .....  | 48 |

## 1. GOALS OF SYSTEM RESEARCH IN THE TANGRAM PROGRAM

The vast quantities of data available on-line presents a tremendous opportunity for organizations to perform large-scale data analysis and information extraction. Research in machine learning and data mining continually produces new and improved algorithms and data analysis capabilities such as feature selection, relational learning, event detection, social networks analysis, and spatial clustering among others. However, the performance of different algorithms varies widely as a function of the characteristics of the data being processed. Furthermore, end-to-end analysis applications demonstrate that algorithms often perform best in combination with others that may enrich the data or prune the hypothesis space. The assembly of such end-to-end applications can take weeks of: (1) manual data selection, integration, and conversions; (2) algorithm selection by experts; (3) manual parameter adjustment of individual algorithms; and (4) manual software integration and execution. Thus, while the composition of multiple algorithms to achieve performance improvement has been shown to be worthwhile and possible in principle, the cost of manually constructing these applications is prohibitive.

The goal of the **System Research (SR)** component of Tangram is to drastically reduce the human effort required to configure and execute new workflows for data analysis from weeks to minutes by eliminating the need for costly human monitoring and intervention. This requires developing robust end-to-end data analysis systems to analyze data from many distributed sources and with many different algorithms and analytical tools. Customized data mining applications need to be automatically assembled by drawing from a library of the best available algorithms. The system needs to assess the computation required by the application and its priority, and execute it efficiently by drawing on computing resources available for execution in a distributed environment. Data providers (called *data catalog* or *DC*) may provide services to access data sources. There can be many organizations playing the role of data providers, and as a result data may be accessible in various catalogs that are in distributed remote locations. Other organizations may provide algorithms, services, models, or implemented codes that can process data and can be used as components of the workflow. We call them *process catalog* or *PC*. These are typically distributed and provided by different organizations. Therefore, an important requirement for workflow systems is that they must rely on distributed services to access the data and algorithms necessary for data analysis. The system needs to incorporate dynamically new algorithms, new data sources, and new computing resources, and learn to adapt its behavior appropriately.

Our ultimate goal is to deliver these capabilities to thousands of users operating in different organizations where data and other resources would be shared. The scale of the computations required is daunting. Our goal is to develop a system that could serve an organization of (on the order of)  $O(10^4)$  users, each issuing  $O(100)$  new and repeated queries daily that may overlap with queries by others, while providing access to  $O(100)$  data sources and databases ranging from  $O(10^9)$  to  $O(10^6)$  records, which are updated almost daily. Such a system requires the ability to: (1) specify, coordinate and prioritize large numbers of complex sequences of analysis steps; (2) scale the amount of processing and storage so as to accommodate ever-increasing sources of data and algorithmic

complexity; and (3) dynamically share data, computing, storage, and analysis resources across communities of information providers, service providers, and users.

The SR team has extensive experience helping scientific communities move from manually intensive and limited settings to distributed computation environments in which complex large-scale applications that compete for shared resources are managed, optimized, executed, and recorded. Earth scientists, physicists, biologists, and astronomers, among others, are able to routinely exercise complex data processing of unprecedented scale with drastically reduced effort and increased computation, compared to capabilities they had a decade ago. We structure these applications as **workflows** described in high-level, declarative notations, and comprising hundreds of steps and processing large quantities of data that comes from multiple, distributed data sources. We use grid computing infrastructure to manage the execution of large numbers of concurrent workflows on shared distributed resources. For this program, we applied these technologies to bring the power of unprecedented scale and synthesis to data analysis problems.

Our approach combines three central ideas:

1. **Workflows** as first class citizens, with rich representations of algorithmic requirements and data products so they can be automatically assembled and executed to respond to user-supplied queries.
2. **Semantic representations** to enable automatic generation of complex workflows and systematic management of many workflow candidates.
3. **Grid computing** to manage the high-performance execution of many workflows, in distributed cross-organization environments.

## 2. ARCHITECTURE OF THE WORKFLOW SYSTEM

The SR workflow system is an extension of the existing *Wings/Pegasus* workflow system. Major extensions under this program include a Workflow Generation system for fully automatic algorithm and data selection, and an Ensemble Manager for dynamic management of concurrent workflow generation and execution.

Given a line of inquiry, one or more *workflow requests* are created. Each workflow request contains a description of desired workflow data products and other constraints such as deadlines for returning an answer. Workflow requests are submitted to the workflow system. A given workflow request results in the execution of several workflows, and the answer is returned from workflow execution. Because many such workflow requests can be submitted to the workflow system concurrently, the workflow system needs to prioritize these requests and assign resources accordingly. The details of workflow generation and execution steps are recorded and are available for inspection after execution.

## 2.1 Major Components of the Workflow System

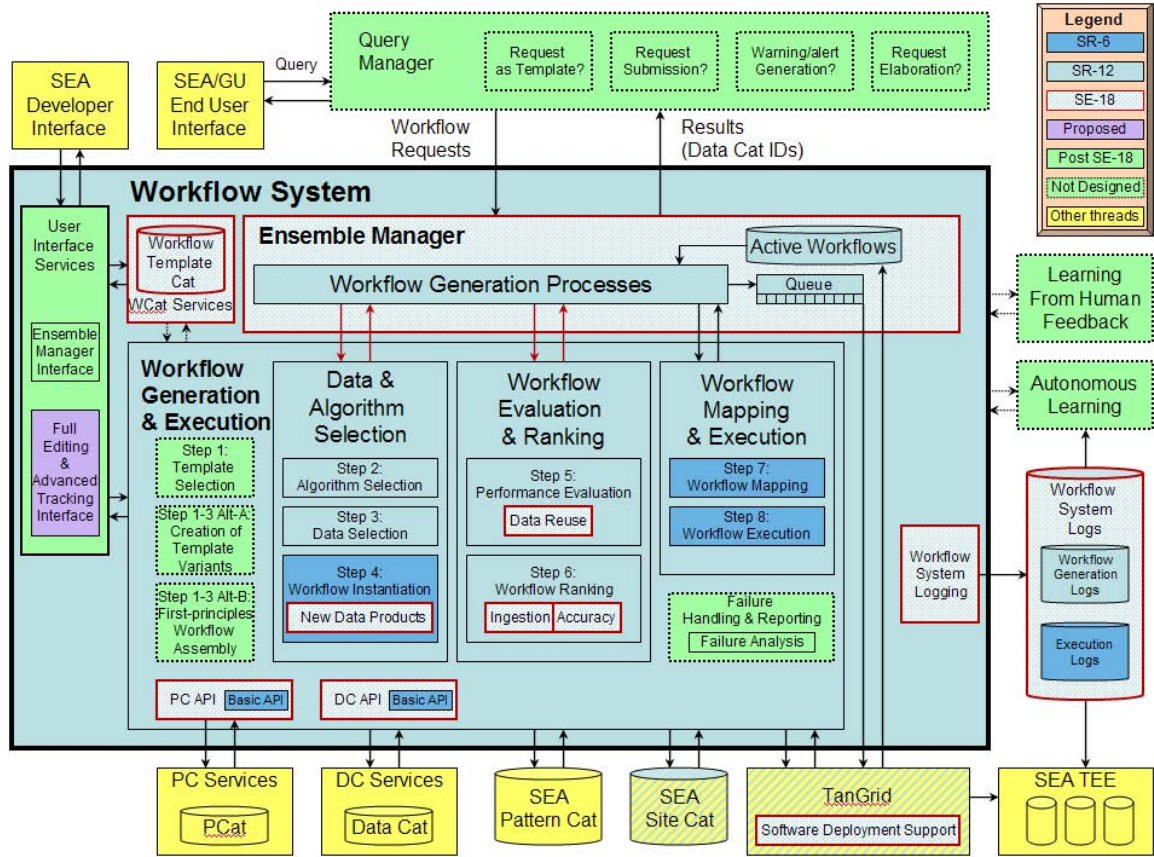


Figure 1: SR Architecture

Figure 1 shows the different components of the workflow system in different stages. Major components of the Workflow System are: 1) Workflow Generation and Execution System 2) Ensemble Manager and 3) Workflow System Logging.

The Workflow Generation and Execution System was developed as an extension of the Wings/Pegasus workflow system. It includes steps for data and algorithm selection, workflow evaluation and ranking, and finally workflow mapping and execution. APIs to DC (data catalog) and PC (process catalog) have evolved from simple basic versions to extended ones that are used for the System Evaluation at 18 months (i.e., SE-18 evaluation).

The *Ensemble Manager (EM)* component of the SR system is responsible for supporting the creation and the execution of multiple workflows at the same time.

Current workflow systems allow only sequential or uncoordinated creation and execution of a single workflow. The Ensemble Manager that we developed coordinates and efficiently handles planning and executing 100's to 1000's of workflows simultaneously on the Grid. The EM manages sets of workflows, with each set specified as a *workflow ensemble*. A workflow ensemble may, for example, contain the pool of candidate workflows being considered for a given step in the workflow generation process. The EM is invoked to perform on workflow ensembles any of the generation, planning, and execution steps of the workflow generation processes.

The Workflow System Logs contains records of the workflow generation and execution process within the SR system. The Workflow System Logs are being used by System Evaluation Architecture (SEA) and other teams. SEA is using the Workflow System Logs to retrieve records of the system's performance for a given workflow request, as well as records of the workflow generation and execution process that can be audited and analyzed. PC is using the Workflow System Logs to learn performance characteristics based on workflow execution data. Many details are available about the Workflow System Logs, including examples and ontologies, as described in Section 6.2.

SR has developed a set of new capabilities for our own Grid called "TanGrid" support, software deployment, and SEA Site Catalog, including providing probes to gather TanGrid site information and generate a dynamic site catalog.

The query manager, template library, and the components for end user interaction were considered as post SE-18 activities.

## 2.2 The Wings/Pegasus Workflow System

The **Wings/Pegasus** workflow system was originally developed under several National Science Foundation grants and has been used in several large-scale distributed scientific applications [Wings 2008; Pegasus 2008; Deelman et al., 2003; Deelman et al., 2005; Deelman et al., 2006; Gil et al., 2005; Gil, 2006; Gil et al., 2006; Gil et al., 2007; Kim et al., 2007; Kim et al., 2006]. It is the basis for the SR Workflow System, which is extending Wings/Pegasus with new capabilities and integration requirements of this program. We summarize here briefly the Wings/Pegasus Workflow System.

Wings uses ontology-based descriptions of software components and data sources to generate a workflow. It includes a workflow template editor that allows a user to define useful and reusable combinations of components and their dataflow. Wings focuses on the domain-dependent aspects of the workflow, but not in execution-level concerns. Wings takes the user's workflow requirements and generates a high-level workflow for Pegasus.

Pegasus generates executable workflows by assigning execution resources to the computations in the workflow. Pegasus also reduces the workflow execution time by eliminating unnecessary computations whose results already exist and can be reused, and reorganizing the structure of the workflow to minimize job queuing time and data movements. It then submits the workflows to the grid for execution and monitors their status and repairs routine low-level execution failures.

To support the creation and validation of very large workflows, Wings/Pegasus takes an approach that considers three stages for workflow creation, where each stage corresponds to a different level of abstraction, and where a new type of information is being added to the workflow. Wings supports the first two layers, while Pegasus supports the third.

The first layer of workflow creation defines workflow templates that are data- and execution-independent specifications of computations. Workflow templates express repetitive computational structures in a compact manner and identify the types of components to be invoked and the data flow among them. A workflow template is an abstract specification of a workflow, with a set of nodes and links where each node is a placeholder for a component or component collections (for iterative execution of a program over a file collection), and each link represents how the input and output parameters are connected. The nature of the components constrains the type of data that the workflow is designed to process, but the specific data to be used are not described in the template. A workflow template can be shared and reused among users performing the same type of analysis.

The second layer of workflow creation uses workflow templates as a starting point to create workflow instances that are execution-independent. Workflow instances specify the input data needed for an analysis in addition to the application components to be used and the data flow among them. A workflow instance can be created by selecting a workflow template that describes the desired type of analysis and binding its data descriptions to specific data to be used. While a workflow instance logically identifies the full analysis, it does not include execution details such as the physical replicas or resources to be used. That is, the same workflow instance can be mapped into different executable workflows that generate exactly the same results but use different resources available in alternative execution environments.

The third and final layer of workflow creation maps workflow instances onto executable workflows. Executable workflows are created by taking workflow instances and assigning actual resources that exist in the execution environment and reassigning them dynamically as the execution unfolds. Executable workflows fully specify the resources available in the execution environment (e.g., physical replicas, sites and hosts, and service instances) that should be used for execution. This is the stage done by Pegasus.

Wings implements the approach outlined above taking a workflow template and initial input file descriptions, and creating a workflow instance called DAX (DAG XML description). Pegasus transforms a DAX into an executable workflow through a mapping that assigns tasks to available grid resources for execution. The details on the representation and the workflow reasoning are reported in [Kim et al., 2006; Gil et al., 2006; Deelman et al., 2004].

Details about how workflows are composed and executed can be found in the Wings/Pegasus Provenance Challenge Site [WingsPegasus 2007]. A detailed example shows how a workflow is created with Wings and Pegasus and includes pointers to the ontologies, component models, and metadata representations for a workflow template, a workflow instance, a DAX, and an executable workflow.

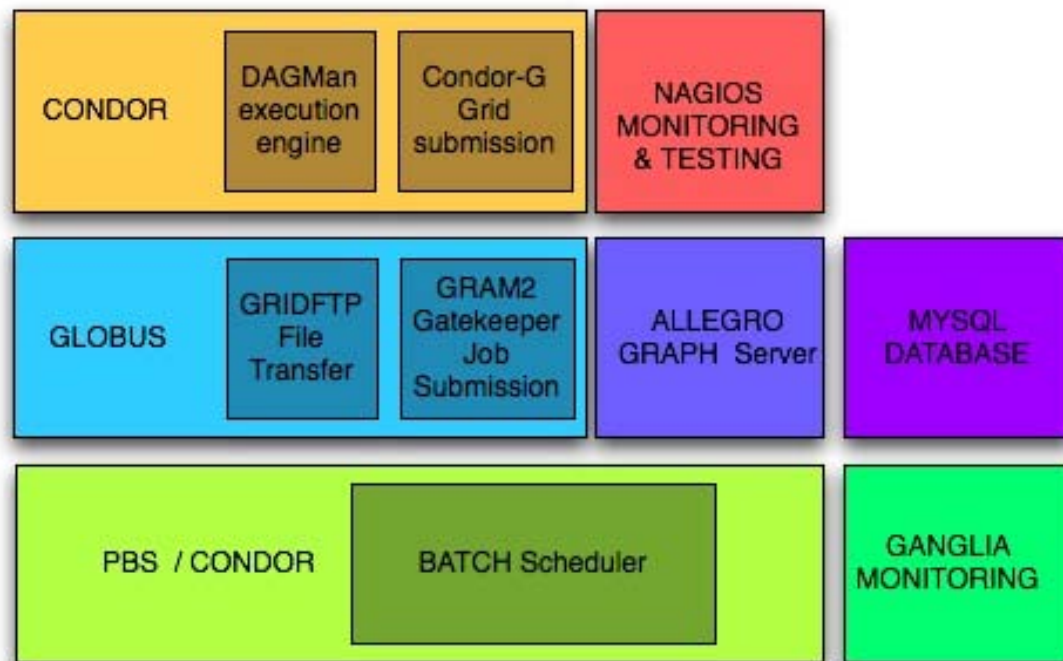
The Wings/Pegasus workflow system was used to conduct the SR-6 demonstration, six months into the program.

## 2.3 Middleware Services Used by the Workflow System

The Workflow System uses the following middleware services shown in Figure 2.

- **Condor Batch System:** Condor is used on the Submit node (<http://cs.wisc.edu/condor>) by the Ensemble Manager to co-ordinate the generation, planning, and execution of multiple requests and workflows across the various nodes of the TanGrid.
- **Globus GRAM2 Gatekeeper:** The Globus Gram2 Gatekeeper and batch scheduler specific jobmanagers (jobmanager-condor or jobmanager-pbss) from the Globus Toolkit (<http://www.globus.org>) are installed on the head nodes of the cluster for allowing job submission from remote machines to the local batch scheduler on the cluster.
- **Globus Gridftp Server:** The Gridftp server from the Globus Toolkit is installed on the head node of a cluster. This server enables high speed and efficient data transfers of large datasets across the TanGrid. This service is used for any kind of data transfer required for execution of the workflows.
- **Local Batch System:** Condor or batch systems are installed on the cluster nodes to allow job scheduling within the cluster. These batch systems enable sharing of the resources by multiple users and support prioritization of jobs, wall-time, etc.
- **Allegro Graph Server:** Franz's Allegro Lisp and Allegro Graph Server products are used for storing the data in triple form in the Allegro Graph Knowledge Base.
- **MySQL Database:** A MySQL relational database management system is used on the TanGrid to store status information generated while executing the Workflows via the Ensemble Manager. Also the MySQL Database maintains a Pattern Registry that is used by Pegasus to retrieve locations of patterns to be used for the workflows.
- **Ganglia Monitoring:** Ganglia Monitoring probes are installed on each node on the cluster which publish monitoring information like load, cpu usage, memory usage, network usage and other such statistics including historical information. This information is aggregated for each cluster and published to a central publishing site at <http://wind.isi.edu/ganglia>.

- **Nagios Monitoring and Testing:** Nagios Monitoring and Testing framework is installed on one of the TanGrid Nodes. Its function is to run various testing probes repeatedly at a fixed interval in order to test the health of the TanGrid nodes and services. The Nagios monitoring page can be seen at <https://artemis.stdc.com/nagios2/>



**Figure 2: Middleware Services for the Workflow System**

## 2.4 Workflow Execution Facilities

SR has provided two clusters at Information Sciences Institute (ISI) to be used as Workflow Execution Facilities. Additionally each group has a Submit node installed at their site. The two clusters at ISI used for execution are described below.

VIZ Cluster:

- 1 Linux Head Node
  - Dual CPU, Dual Core Xeon 2.4Ghz (32 bit)
  - 2GB Memory
  - Gigabit Ethernet
  - 150Gbs of shared storage
  - Debian 3.1 Linux
  - PBS Batch System
- 8 Linux Cluster Nodes each with
  - Dual CPU, Dual Core Xeon 2.4Ghz (32 bit)

- 2GB Memory
- Gigabit Ethernet
- 150Gbs of shared storage
- Debian 3.1 Linux

WIND Cluster:

- 1 Linux Head Node
  - Dual CPU, Dual Core Xeon 2.3 Ghz (64 bit)
  - 8 GB Memory
  - Gig Ethernet
  - 7Tb of shared storage
  - Fedora Core 7
  - Condor Batch System
- 5 Linux Cluster Node
  - Dual CPU, Dual Core Xeon 2.13 Ghz (64 bit)
  - 4 GB Memory
  - Gig Ethernet
  - 7Tb of shared storage
  - Fedora Core 7
- 1 Windows Cluster Node
  - Pentium 4 2 Ghz (32 bit)
  - 1 GB Memory
  - 100Mbps Ethernet
  - Windows Xp SP2

### 3. RESEARCH RESULTS

SR's work under this program covered three major research areas: automatic workflow generation, workflow ranking, and managing execution of multiple workflows.

#### 3.1 Automatic Workflow Generation

Computational workflows are a powerful paradigm to represent and manage complex applications, particularly in large-scale distributed data analysis. Workflows represent application components that result in individual computations as well as their interdependencies in terms of data flow. Workflow systems use these representations to manage various aspects of workflow creation and execution for users, such as the automatic assignment of execution resources.

SR's research on workflow generation provides an approach to automating a new aspect of the process: the selection of application components and data sources. We present a formalization of the problem and an algorithm that elaborates the high-level template into a set of fully ground workflows with specific choices of data sources and codes to be used so that they can be submitted for mapping and execution. The algorithm starts from a user-specified request that includes a high-level workflow template and any

additional constraints on results or data sources. It searches through the space of possible candidate workflows by creating increasingly more specialized versions of the original template and eliminating candidates that violate constraints as components and data sources are selected.

Figure 3 shows an overview of the distinct stages during workflow generation. A pool of workflow candidates are formed from the initial request. Each stage adds increasingly more detail to each candidate workflow until they are ready to be submitted to the workflow mapping and execution engine. In this process candidate workflows can be added or eliminated. If at any point there are no workflow candidates remaining, the algorithm ends returning an empty result.

The initial request is assumed to contain template/seed pairs that are each well-formed and unified with the template variables. In the first stage, a seeded workflow is created from each template/seed pair by merging the seed with the workflow template constraints. These seeded workflows are considered to be the initial pool of candidate workflows. The next stage propagates constraints from the workflow outputs to the workflow inputs to create binding-ready workflows. Next, input data sources that satisfy the constraints imposed by the workflow are found to create a pool of candidate bound workflows. In the next stage, the properties of input data sources are propagated through each component, resulting in configured workflows. Finally, unique identifiers for workflow data products are obtained to create workflow instances, and specific command invocations are associated with each workflow component to create ground workflows. Finally, the candidate workflows are ranked and the k-best candidates are submitted to the workflow mapping and execution engine. The detailed individual steps are described in Section 3.1.3.

Our algorithms assume a distributed architecture where data and process (component) catalogs are separate from the workflow system. To function in such a distributed architecture, the algorithm explicitly poses queries to external catalogs, and therefore any reasoning regarding data or component properties is not assumed to occur within the workflow system. To illustrate our approach, for simplicity, we use workflows composed of machine learning algorithms as components from the well-known Weka library and datasets from the widely-used Irvine repository. We also show our implementation using the W3C Web Ontology Language (OWL) and associated reasoners to implement the workflow system as well as the data and process catalogs. This research demonstrates the use of artificial intelligence techniques to support the kinds of automation envisioned by the intelligence community for large-scale distributed data analysis.

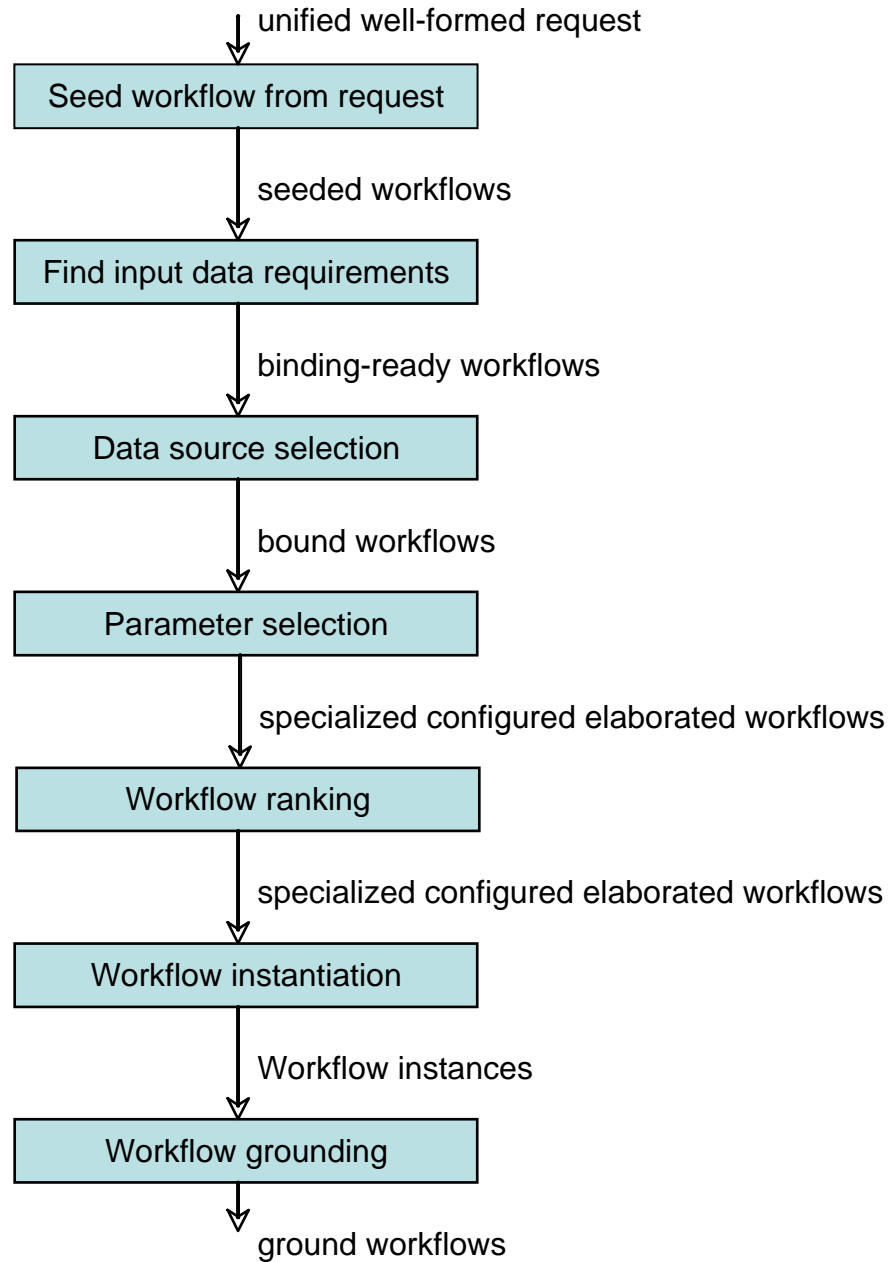
### **3.1.1 Data, Components, Workflows, and Workflow Requests: Requirements and Examples**

#### Data

Data objects have *metadata properties*, which are used to describe useful features of the data. An example metadata property is the size of a datasets, and whether the data is continuous or discrete. Metadata properties can specify the type of the data, for example

whether they are instances or models, and for models whether they are a decision tree model or a Bayes model.

---



**Figure 3: Stages During Workflow Generation**

**Table 1: Representation of Abstract and Concrete Workflow Components**

| (a) Abstract Component   | (b) Concrete Components                                       |   |
|--|---|---|
| Component ID: DecTreeModeler<br>is Abstract                          | Component ID: ID3-Modeler<br>is DecTreeModeler<br>is Concrete | Component ID: Lmt-Modeler<br>is DecTreeModeler<br>is Concrete |
| Input: d: Dataset<br>hasSize s:Size                                  | Input: d: Dataset is <b>Discrete</b>                          | Input: d: Dataset is <b>NoMissingVals</b>                     |
| Params: i: ClassIndex<br>j: maxJavaHeapSize<br>j <- 256x rem(s/1000) | Params: i: ClassIndex<br>j: maxJavaHeapSize                   | Params: i: ClassIndex<br>j: maxJavaHeapSize                   |
| Output: o: Model is DecTree  | Output: o: DecTree-Model                                      | Output: o: DecTree-Model                                      |

Concrete components correspond to executable codes, abstract components correspond to classes of components with common general properties. An important requirement is the ability to describe new workflow data products since we need to be able to refer to data objects that do not yet exist and that will only exist once the workflow is executed. In addition, we need to anticipate the metadata properties of those objects. We refer to this as the *projected metadata* for a data object. For example, if the training dataset is of a particular kind or domain, such as weather data, then we can state that the model learned is also for weather data and that the classifier is expected to operate on test data that are also weather data. To support this, we need to be able to state not only that the domain of the training data (e.g., weather) must be the same as the domain for the given test data, but also to state or infer that the learned model is of that same domain (e.g., also weather). Notice that the projected metadata for a data object may be different than the actual metadata obtained once the data object is created. For example, we can anticipate that a learned model for a training set of 1,000 instances will have 25 rules, but the actual learned model may end up containing exactly 26 rules. The data catalogs need to handle the fact that these new projected data products may never materialize, either because the workflow will not be selected for execution (in favor of other alternatives) or because the execution of the workflow may fail.

Representing workflows requires being able to represent statements about data objects that include the ability to:

- 1) refer to the data objects that will be used as input data,
- 2) attach properties to those data objects,
- 3) state relations among properties of different data objects,
- 4) refer to new workflow data products and their properties

### Components

Components often require representing properties of their input or output data. For example, that a component that discretizes a dataset has as output a dataset that is discrete (instead of continuous data). Components have *argument identifiers* that enable the workflow system to refer to particular arguments of the component. For example, the component ID3-Modeler has an input dataset whose identifier is “d”.

Components in a component catalog may be *abstract* or *concrete*. Concrete components model pieces of software that can actually be executed while abstract

components are descriptions of the common features of a set of concrete components, in a similar sense to the way an abstract class in object-oriented programming gathers the commonalities of its subclasses. Table 1(a) shows an example of an abstract component, which represents all common properties of decision tree modelers such as the output is in the form of a decision tree. Table 1(b) shows two examples of concrete components. The ID3-Modeler requires input data that are discrete, while the LMT-Modeler requires the training examples used as input to have no missing values for their features. The abstract component illustrates how the value of a metadata property of an input dataset (s) is used to set the value of a parameter (h).

We have the following requirements for representing components:

- 1) represent input data, parameters, and output data in each with a unique argument identifier,
- 2) represent constraints on the values that arguments can take, including type,
- 3) represent constraints across argument values,
- 4) represent classes of components based on common properties, and
- 5) ability to generate an appropriate invocation command.

## Workflows

Workflows have complementary representations of *structure* and *constraints*. The structure of a workflow reflects the dataflow among components, while the constraints reflect interactions among components and datasets. We explain now both aspects of the representation in more detail.

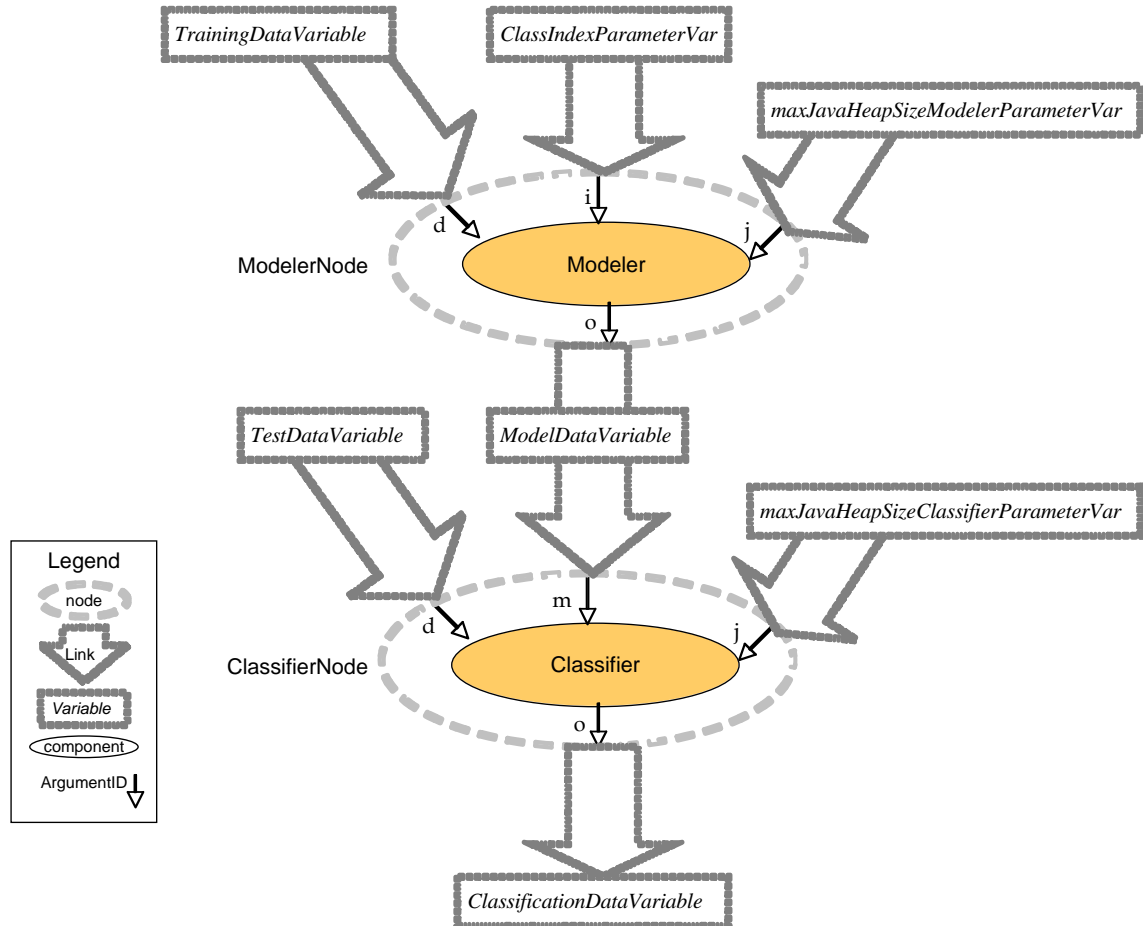
The structure of a workflow is specified as a set of *nodes*, each corresponding to a component, and a set of *links* that reflect the dataflow across components. For simplicity, we use links to specify input and output data and refer to them as input and output links, with an empty origin node and an empty destination node respectively. Similarly, we use parameter links for parameters, and give them an empty origin node. All other links are in-out links and include both an origin and a destination node.

Figure 4 shows a representation of a workflow. The dotted ovals and the dotted lines represent workflow nodes and links respectively. The ModelerThenClassifier workflow consists of two nodes (modelerNode with Modeler component and ClassifierNode with Classifier component), two input links (for modelerTrainingData and classifierInputData), two parameter links (for javaMaxHeapSize and modelerClassIndex), one in-out link (for outputModel and classifierInputModel), and one output link (for classifierOutput). In this workflow, there are several data variables shown. One is TrainingDataVariable, which refers to the input data that will be used as training data and can be bound to any data sets available. Another data variable is ModelDataVariable, which refers to the data product generated after the Modeler component will be executed. The workflow in the figure is variabilized, and its input data variables can be bound to many possible combinations of data inputs. Links specify which argument identifier of the component is associated with the link. In Figure 4, the argument identifiers are shown next to the solid arrows inside the nodes. For example, the learned model corresponds to the “m” argument id of the Classifier component. We include here a parameter that is used in the Weka implementation and that specifies the allocation of memory to be used (indicated with a “j” argument identifier on both components) and should be set in proportion to the size of

the input data sets. We will use this parameter in later sections to illustrate how the components can be automatically configured during workflow generation.

---

**Workflow Structure:**



**Figure 4: The representation of a Workflow to learn a model and then classify data.**

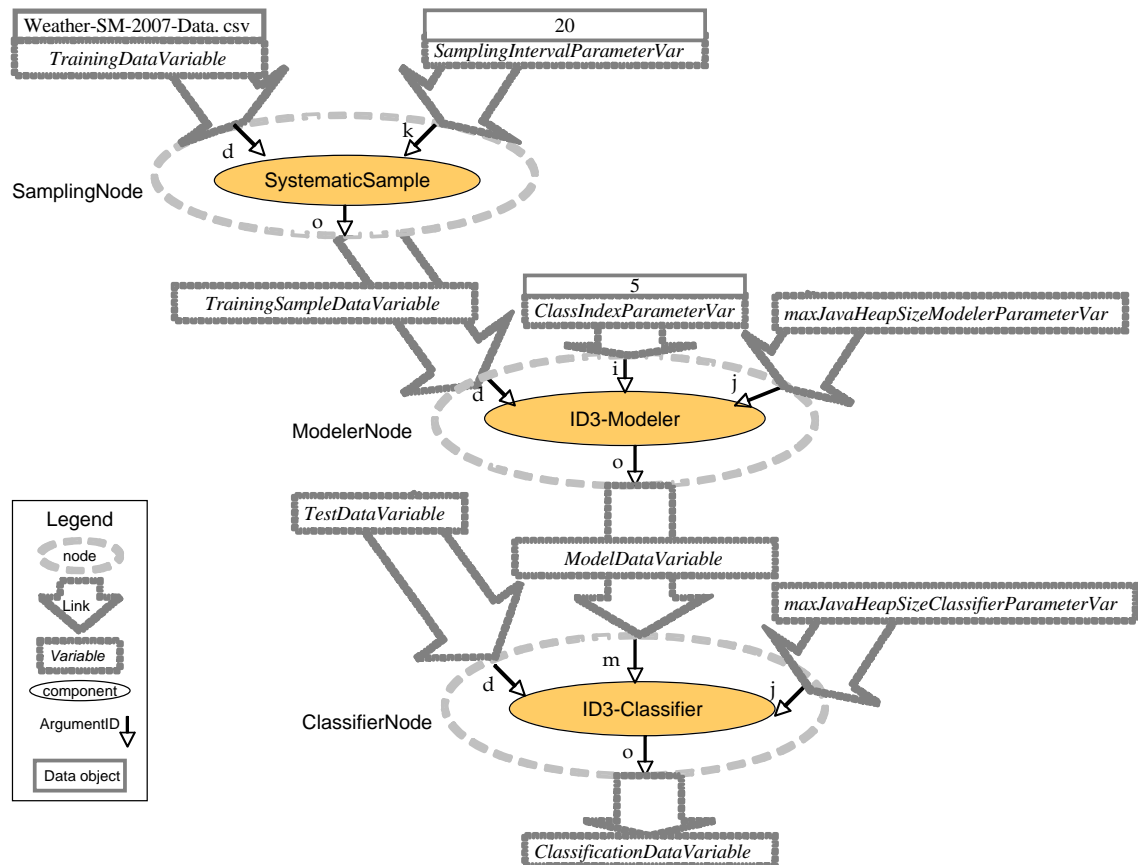
**Workflow Constraints:**

TrainingDataVariable  $\neq$  TestDataVariable  
 Domain of TrainingDataVariable = Domain of TestDataVariable

The nodes are shown in dashed ovals, links are shown in dotted arrows. The top illustrates the representation of the structure of the workflow, the bottom shows the constraints of the workflow.

---

### Workflow Structure:



**Figure 5: A workflow to learn a model to predict weather**

### Workflow Constraints:

TestDataVariable has Domain = weather  
TrainingDataVariable = Weather-SM-2007-Data.csv  
SamplingIntervalParameterVar = 20  
ClassIndexParameterVar = 5  
TrainingDataVariable  $\neq$  TestDataVariable  
Domain of TrainingDataVariable = Domain of TestDataVariable

The training data is sampled first at a set rate. The workflow constraints specify bindings of data variables as well as parameter settings. Workflow constraints also express a restriction on the test dataset that it contains weather data, which avoids incorrect use of the workflow with other kinds of data.

In our work, the structure of a workflow is constrained to be a directed acyclic graph (DAG). This is a very simple structure that we have found very useful in many fronts: including hiding programming constructs from users, facilitating reasoning about workflows (in particular automatic workflow generation), and last but not least recovery of execution when a job fails in the middle of the workflow. Many workflow languages depart from this basic structure and enable constructs such as conditionals and iterations through global variables. Using DAGs, we are able to support simple forms of iteration over data collections, as well as conditional execution based on data types [Gil et al 07a]. We have found this structure to be very manageable and to cover what was needed for a wide range of applications.

Figure 5 shows another example of a workflow customized to learn to predict weather data. This is a workflow that has data objects assigned to some of the data variables and values assigned to some of the parameters, both done through the workflow constraints. This workflow also illustrates how constraints represent additional metadata properties of the input data. In this case, a constraint indicates that the domain of the test dataset used must be weather. This constraint will avoid the incorrect use of this workflow to make predictions over non-weather data.

The representation of the structure of the workflow is essentially syntactic in nature, as it is concerned with having a complete specification of the direction of the dataflow for all the inputs and outputs of the components. The representation of the constraints is semantic in nature, and is concerned with having a consistent specification of the nature of the data exchanged among components through the dataflow.

In summary, to represent workflows we have the following requirements:

- 1) represent dataflow across components,
- 2) represent data variables that are generic placeholders for actual datasets, so we can have reusable workflow templates,
- 3) represent constraints on data variables,
- 4) represent constraints across data variables, and
- 5) represent different degrees of generality in the workflows, including bindings for input datasets and values for parameters.

### Workflow Requests

Our goal is to automatically generate responses for a variety of requests from users by generating and executing workflows that satisfy the requests. Users may specify a variety of criteria in the requests, such as:

- *Functional properties*: Users often want to use workflows based on the nature of the computations performed or the desired data products. These include:
  - *component-centered properties* that refer to the kinds of computations performed by the workflow components

- *data-centered properties* that refer to desired data products, or that specify that a certain type of data of interest to the user must be used in the workflow.

Example of output data properties: Create a Naive Bayes model of labor data.

Example of input data properties: Classify iris data using a naive-bayes model with three classification classes and created from at least 500 instances.

Example of component properties: Create a model of labor data using ID3.

Example of component properties: Create a model of labor data with no sampling steps in the workflow (i.e., using the complete training data set).

- *Structural properties*: Users may provide constraints on the structural composition of the workflow concerning the relative ordering of steps. For example, a user may seek a workflow that performs data aggregation on a collection of datasets before performing clustering operations.

Example: Sample soybean data and then create a Naive Bayes model.

Example: Use ModelThenClassify workflow with soybean data.

- *Non-functional properties*: These properties express user requirements regarding workflow performance and other costs. We highlight two here:
  - Execution time: A desired turnaround time for obtaining results.
  - Result quality: A threshold of quality or accuracy measured in some domain-relevant metric.

Because some of these requirements may be in conflict, users may state additional combination functions or preferences. For example, there is typically a tradeoff between execution time and result quality, where shorter time often implies a lower quality results. A combination function may be expressed in the request when both time and quality matter.

Example: Create a model of soybean data with maximum accuracy.

Example: Classify iris data and minimize the response time.

- *Resource properties*: Users may have specific requirements about the execution resources to be used in executing the workflow. For example, for a workflow designed to compare the performance of a set of algorithms the user may request that all the algorithms may be executed on the same target architecture, or that the datasets used should be those existing at specific locations. Users may also request that specific resources should not be used, such as datasets generated by prior workflow executions or datasets that have not been updated for some period of time.
- *Cumulative properties*: These are properties of workflows that are derived through usage. Users may prefer to use workflows that are most frequently used by a user group, or more popular for a given function.
 

Example: Create a model for soybean data using the most popular decision tree modeler.
- *Comparative properties*: Properties that are derived by comparing across possible candidate workflows. These can be used as ranking functions that drive the selection of workflows that have higher ranking.
 

Example: Create a model for soybean data with minimum description length.

In our work, users always specify a workflow template to be used in combination with a *seed* that specifies additional constraints. The seed can specify constraints on the inputs, outputs or intermediate data variables, as well as requirements on particular

components, data objects or parameter values to be used. The workflow template essentially specifies functional or structural requirements that users have. In general, scientific applications and data analysis applications are run with a specific workflow structure or template in mind [Gil 06].

An example of a request is one to create a classification of a weather data object (i.e. the domain of the classifierOutput is weather) using the ModelerThenClassifier workflow (shown in Figure 4), with a value of 5 for the “i” parameter of the Modeler step.

If a workflow request did not include a workflow template, the seed can be used to search for relevant workflows in the library that could be used to accomplish the request. Matching requests and workflow templates is a unification problem [Baader and Narendran 01]. There is a large body of work on matching in the case-based reasoning literature [Ashley and Aleven 97; Bergmann and Stahl 98; Champin and Solnon 03; Forbus et al 94] as well as in matching in first-order and in description logic [Li and Horrocks 03; Baader et al 05; Hull et al 06]. Graph matching techniques have been used to retrieve workflows based on structural properties [Goderis et al 06]. End users, however, will most often formulate their requests based on domain-relevant features of the workflow rather than referring to the implementation details of the software artifacts described in our current workflow representations. Therefore, ideally each workflow and data product would be described in terms of domain-relevant models in addition to software-level descriptions.

Another alternative to responding to a workflow request that does not include a template is to use a generative search algorithm to generate a workflow by selecting one component at a time using the seed as a goal statement [Blythe et al 04a; Blythe et al 04b]. However, those approaches require very detailed models of the components in order to support composition. We assume that a workflow request is formed by a workflow template and a seed that specifies additional constraints.

In summary, in order to satisfy the requirements in user requests, we need to be able to:

- 1) represent properties of desired input and output data,
- 2) merge the retrieved templates with the requirements expressed in the request , and
- 3) propagate the effects that the requirements expressed in the request pose on other components and datasets of the workflows being considered.

### 3.1.2 Formalization

This section shows a formalization of our framework to satisfy the above requirements. The formalization describes how the workflow system can model data, components, and workflows. We also show the basic functions that the workflow system needs in order to access external data, component, and workflow catalogs. Although our own implementation uses a particular representation formalism, we use a more general formalization here using algebraic specification [Liskov and Guttag 86]. We show later our implementation of this formalization.

## Data and Data Catalogs: Formalization

To refer to data objects relevant to a workflow we use the term *data entities*. Data entities can be an existing initial dataset, a workflow data variable, a component parameter, and a specific workflow data product.

We use the term *data object description* (DOD) to refer to a set of metadata annotations that describe the properties of a given data object. DODs for input data variables effectively constrain the possible input data that can be used to bind the data variables standing for data objects. DODs for component parameters can be used to constrain the values that the parameter can take, and to specify the value of a component parameter.

In our work we use a particular representation formalism (first-order logic). However, we wish to make our framework very general and applicable to alternative formalisms. Therefore, without actually committing to any formalism for DODs, we assume that the chosen formalism allows writing literal annotations and literal relational annotations as defined below.

Formally, we assume:

- **E** as the set of data entities in the workflow,
- **P** as the set of metadata properties,
- $\tau(p)$  is the set of possible values for a metadata property  $p \in P$
- **V** as the set of possible values of the component parameters
- **R** as the set of relations between pairs of property values,

Given **E**, **P**, and **R**, we assume that the DOD formalism allows expressing:

- *literal DODs* to specify the value of a metadata property for a given object as:  
 $\langle e, p, v \rangle$

where  $e \in E, p \in P, v \in \tau(p)$ .

Example: For the workflow shown in Figure 5,  $\langle \text{TestDataVariable}, \text{has Domain}, \text{weather} \rangle$

- *literal relational DODs* to specify a relation between the values of two properties of the same or different data objects as:

$\langle r, e_1, [p_{11}, \dots, p_{1n}], e_2, [p_{21}, \dots, p_{2m}] \rangle$

where  $e_1, e_2 \in E, p_{11}, \dots, p_{1n}, p_{21}, \dots, p_{2m} \in P, r \in R$ , and  $n, m \geq 1$

Representing that relation  $r$  holds between the value at the end of the property chain  $p_{11}, \dots, p_{1n}$  starting in  $e_1$  and the value at the end of the property chain  $p_{21}, \dots, p_{2m}$  starting in  $e_2$ . Formally:

$\exists x_1, \dots, x_n, y_1, \dots, y_m$  such that

$\langle e_1, p_{11}, x_1 \rangle \wedge \dots \wedge \langle x_{n-1}, p_{1n}, x_n \rangle \wedge \langle e_2, p_{21}, y_1 \rangle \wedge \dots \wedge \langle y_{m-1}, p_{2m}, y_m \rangle$  and

$r(x_n, y_m)$

Example: To add a restriction in the workflow in Figure 5 that it can only be used to make predictions with weather data that is from the same county as the training data, we can state:

<equals, TestDataVariable, [ has-area, has-county], TrainingDataVariable, [has-area, has-county]>

which represents that:

< TestDataVariable, has-area, A1>

<A1, has-county, C1>

< TrainingDataVariable has-area A2>

<A2, has-county, C2>

<C1, equals, C2>

With these representations for DODs, we can represent workflow constraints. For example, the workflow constraints shown in Figure 5 can be expressed as the set of DODs:

M3= {<TestDataVariable, hasDomain, weather>  
       <TrainingDataVariable, has-value, Weather-SM-2007-Data >  
       < SamplingIntervalParameterVar, has-value, 20 >  
     < ClassIndexParameterVar, has-value, 5>  
     <not-equal, TrainingDataVariable, [has-value], TestDataVariable, [has-value]>}

We denote as  $M(E)$  a set of metadata annotations that only refer to entities in the set  $E$  and as  $E(M)$  the set of entities referred in the set  $M$  of annotations. We now describe the functions that implement this notation. We assume that the formalism for metadata allows us to build a function that implements  $M(E)$  to retrieve the subset of DODs that only refer to a set of given entities  $E$ , and a function that implements  $E(M)$  as follows:

— **entity-DODs:**  $E \rightarrow M$

which returns a subset of the given set of DODs that only refer to the given set of entities. Formally, if  $MA$  is a set of DODs:

entity-DODs(  $V, MA$  )  $\subseteq MA$ , and  $E(\text{entity-DODs}( V, MA )) \subseteq V$

Example: entity-DODs ( {TestDataVariable TrainingDataVariable}, M3)  $\equiv$

{<TestDataVariable, hasDomain, weather>  
       <TrainingDataVariable, has-value, Weather-SM-2007-Data >  
       <not-equal, TrainingDataVariable, [has-value], TestDataVariable, [has-value]>}

— **get-entities-in-DODs:**  $M \rightarrow E$

Example: get-entities-in-DODs(M3)  $\equiv$  { TestDataVariable TrainingDataVariable  
       SamplingIntervalParameterVar ClassIndexParameterVar }

Note that  $M$  and  $E$  are the types of all DODs and all entities respectively, and  $V$  and  $MA$  are the values for a specific call to the function.

With these definitions, we can now formalize the functions that a data catalog must be able to support. A data catalog must include functions to retrieve data objects given their DODs. In addition, it must provide a function to combine sets of DODs and to warn when they are not possible to combine (we will illustrate the need for this in the workflow generation algorithm below).

Given a set  $D$  of data object identifiers contained in the data catalog  $DC$ , and denoting as  $Vars$  a set of workflow data variables that can be annotated with DODs, we can define the following functions:

- **obtain-DODs:**  $D \rightarrow M$   
Return as a DOD all the metadata properties and values of the given data object identifier.
- **assign-identifier:**  $M \rightarrow D$   
Assigns a unique identifier based on a given set of metadata properties and values.
- **assert-predicted-DODs:**  $DC\ M\ D \rightarrow DC$   
Register in the data catalog all the predicted metadata properties and values of the given data object identifier.
- **find-data-objects:**  $M(Vars) \rightarrow \{ \{ \langle Vars \times D \rangle \} \}$   
Given an input set of DODs for several data variables, return a (possibly empty) set of data objects for all the variables in the input DODs that are consistent with the DODs. Each tuple of the form  $\langle Vars \times D \rangle$  is a *binding* for a workflow data variable, where a variable is bound to a data object identifier.
- **combine-DODs:**  $M(Vars)\ M(Vars) \rightarrow M(Vars)$   
Return a set of DODs which combines the metadata properties of two given sets, all of them on a given set of variables.  
In order to be valid, an invocation to combine-DODs must verify that the sets are consistent.

### Components and Component Catalogs: Formalization

In order to support the workflow selection and execution process, a component catalog may include different functions to extract knowledge about the components out of the catalog.

We assume:

- a set  $C$  of components in the catalog. We refer to  $AC$  and  $CC$  as the disjoint subsets of  $C$  for abstract and concrete components respectively.
- a set  $I$  of unique identifiers for each argument of the components (input data, input parameters, and output data)

We can define the following basic functions for the component catalog:

- **inputs:**  $C \rightarrow I$   
Return the identifiers for the input data objects of a component.
- **parameters:**  $C \rightarrow I$   
Return the identifiers for the parameters of a component.
- **outputs:**  $C \rightarrow I$   
Return the identifiers for the outputs of a component.
- **args:** We denote as  $args(c)$  the set of *arguments* of a component  $c \in C$ :

$$\text{args}(c) \equiv \text{inputs}(c) \cup \text{parameters}(c) \cup \text{outputs}(c)$$

- **invocation-command**:  $CC \langle D \times \dots \times D \rangle \langle V \times \dots \times V \rangle \rightarrow \text{String}$   
Return the invocation command for a concrete component, given the values (data object identifiers) for its inputs and the values for its parameters.

When the component library supports these functions for a component, and the invocation command results in a successful invocation and execution of the component, we consider the library to contain a *basic component encapsulation*. Supporting this basic encapsulation of the underlying code does not require a component library to include semantic constraints or properties of data or components.

When a component catalog includes both abstract and concrete components, it needs to support the following functions:

- **is-concrete**:  $C \rightarrow \text{Bool}$   
Determine whether a component  $c$  is abstract or concrete.
- **specialize**:  $AC \ M(I) \rightarrow \{ C \}$   
Return a (possibly empty) set of abstract or concrete components that can specialize a given abstract component using a given set of DODs on the abstract component arguments. We assume that there is a one-to-one mapping between the arguments of each of the concrete components returned and the arguments of the abstract one. When this is not the case, there must be provisions for extending the workflow to account for the additional arguments, possibly to link them to data variables in the workflow, and possibly to add workflow components to the workflow to generate some of the additional arguments.  
In order to be valid, an invocation of  $\text{specialize}(c, M)$  must verify that DODs in  $M$  only refer to the arguments of  $c$ :  $E(M) \subseteq \text{args}(c)$ .
- **specialize-to-concrete**:  $AC \ M(I) \rightarrow \{ CC \}$

Similar to `specialize` but returns only concrete components rather than subclasses.

The next functions support the automatic setting of parameters for a given component. When the component library supports these functions, we refer to the component as *self-configurable*. The functions are defined as follows:

- **is-configurable**:  $C \ M(I) \rightarrow \text{Bool}$   
Determine whether the parameter values for the component can be obtained from the component catalog given a set of DODs on the component arguments. Notice that this does not set the values of any parameters, it simply checks that they can be set by the component catalog.
- **configure**:  $C \ M(I) \rightarrow \{ \langle V \times \dots \times V \rangle \}$   
Return a set of tuples, each tuple specifying values for all the parameters of a component  $c$  given a set of DODs on the component arguments (inputs, outputs and parameters).

In order to be valid, an invocation of  $\text{configure}(c, M)$  must first verify that:

- DODs in  $M$  only refer to the arguments of  $c$ :  $E(M) \subseteq \text{args}(c)$ , and
- $\text{is-configurable}(c, M)$  returns true.

- **is-configured:**  $C \ M(I) \rightarrow Bool$   
Determine whether a component  $c$  is fully configurable based on a given set of DODs on the component arguments.

We define two additional functions that component catalogs can provide in order to support workflow generation:

- **find-DODs-given-output-requirements:**  $CC \ M(I) \rightarrow \{ M(I) \}$   
Return additional metadata properties on the concrete component arguments using the given DODs.

In order to be valid, an invocation `find-DODs-given-output-requirements (c, M)` must comply with:

- DODs in  $M$  only refer to the arguments of  $c$ :

$$E(M) \subseteq \text{args}(c)$$

- $M$  includes some annotations on the outputs of  $c$ :

$$E(M) \cap \text{outputs}(c) \neq \emptyset$$

Ex: `find-DODs-given-output-requirements(ID3-Classifier, <ID3-Classifier-o hasDomain Weather>)`  
 $\rightarrow \{ < ID3-Classifier-d hasDomain Weather> \}$

- **predict-DODs-given-input-requirements:**  $CC \ M(I) \rightarrow M(I)$   
Return additional metadata properties on the concrete component arguments using the given DODs.

In order to be valid, an invocation `predict-DODs-given-input-requirements (c, M)` must comply with:

- DODs in  $M$  only refer to the arguments of  $c$ :

$$E(M) \subseteq \text{args}(c)$$

- $M$  includes some annotations on the inputs of  $c$ :

$$E(M) \cap \text{inputs}(c) \neq \emptyset$$

Ex: `predict-DODs-given-input-requirements(ID3-Classifier, <ID3-Classifier-d has-value weather-2007-31-101501>)`  
 $\rightarrow \{ < ID3-Classifier-d number-of-instances 100>, < ID3-Classifier-j has-value "512M"> \}$

When a component library supports the first function for a given component, we refer to the component as *backward-enabled*. The function is used to propagate through the workflow structure any constraints required from the output data products. When the second function is supported, we refer to the component as *forward-enabled*. This function is used to propagate through the workflow structure any properties of the input data. We also have functions to check whether the component representations in the catalog support these capabilities as follows:

- **is-backward-enabled:**  $C \rightarrow Bool$   
The function **find-DODs-given-output-requirements** is defined for the component.
- **is-forward-enabled:**  $C \rightarrow Bool$

The function **predict-DODs-given-input-requirements** is defined for the component.

Finally, a function to estimate the performance of a component:

— **estimate-performance**:  $C \times M(I) \times \langle V \times \dots \times V \rangle \times A \rightarrow T$

Return estimated performance  $T$  as time of execution for the component for the given metadata and parameter values. The performance is estimated for a given reference architecture  $A$ .

In order to be valid, an invocation estimate-performance  $(c, M, v)$  must comply with:

- DODs in  $M$  only refer to the arguments of  $c$ :

$$E(M) \subseteq \text{args}(c)$$

- $M$  includes some annotations on the inputs of  $c$ :

$$E(M) \cap \text{inputs}(c) \neq \emptyset$$

- $\text{is-concrete}(c)$  returns true
- $\text{is-configured}(c, M)$  returns true

Later, we will show how all these functions are used during workflow generation.

## Workflows

Given a component catalog which includes a set  $C$  of components, a data catalog with a set  $D$  of data object identifiers, a set  $V$  of possible values for component parameters and a set  $P$  of metadata properties, a workflow  $w$  is defined as a tuple of nodes, component to node mappings, data variables, parameter variables, DODs on data and parameter variables, data links, parameter links, data bindings, and parameter bindings. Formally, a workflow is specified as a tuple:

$$\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$$

where:

- $N_w$  is the set of *nodes* in the workflow,
- $\sigma_w$  is a mapping function that associate a component to a node:

$$\sigma_w : N_w \rightarrow C$$

Note that different nodes in the workflow may have the same associated component.

- $DV_w$  is the set of data variables in the workflow
- $PV_w$  is the set of parameter variables in the workflow.
- $M_w$ , is DODs on the data and parameter variables of the workflow:  $E(M_w) \subseteq DV_w \cup PV_w$
- $L_w$ , is the set of links in the workflow. A link  $l$  is represented as a tuple of the form:

$$\langle n_o, o, v, n_d, i \rangle$$

where  $n_o, n_d \in N_w \cup \{\perp\}$ ,  $o \in \text{outputs}(\sigma(n_o)) \cup \{\perp\}$ ,  $v \in DV_w \cup PV_w$ , and  $i \in \text{inputs}(\sigma(n_d)) \cup \{\perp\}$ .

We refer to  $n_o$  as the origin and to  $n_d$  as the destination.

An *input link* to the workflow is one without origin that connects a data variable to an input argument of a component:  $\langle \perp, \perp, v, n_d, i \rangle$   $v \in DV_w$ , while an *output link* is one without destination that connects an output argument to a data variable:  $\langle n_o, o, v, \perp, \perp \rangle$   $v \in DV_w$ .

- $PL_w$ , is the set of parameter links in the workflow. A parameter link  $pl$  is represented as a tuple of the form:

$$\langle pv, n, p \rangle$$

where  $pv \in PV_w$ ,  $n \in N_w$ , and  $p \in \text{parameters}(\sigma_w(n))$ .

- $DVB_w$  is a, possibly empty, set of bindings of the data variables to data object identifiers:

$$\langle dv, d \rangle \equiv \langle dv, \text{has-value}, d \rangle$$

where  $dv \in DV_w$ ,  $d \in D$

- $PVB_w$  is a, possibly empty, set of bindings of the parameter variables to values:

$$\langle pv, v \rangle \equiv \langle pv, \text{has-value}, v \rangle$$

$$pv \in PV_w, v \in V$$

The parameter link indicates which parameter variable in the component corresponds to the link. No values are set to parameters in parameter links, instead the values are set through the parameter bindings.

Workflow catalogs should contain workflows that comply with the basic component encapsulation requirements for all of its nodes' components. That is, all the arguments and argument identifiers specified in the workflow nodes and links have a one-to-one correspondence with the arguments and argument identifiers defined for the nodes' components. We refer to such workflows as *well-formed* workflows. This is a syntactic property concerning the structure of the workflows, and does not concern the constraints or properties that may be defined for data variables or data objects.

We define the following types of workflows:

- *Specialized workflow*:

A workflow which contains only concrete components.

Formally, a workflow  $\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$  where

- $\forall c \in C_w : \text{isConcrete}(c)$

where  $C_w$  denotes the set of components in the workflow:

$$C_w \equiv \{ c \in C \mid \exists n \in N_w : \sigma_w(n) = c \}$$

- *Bound workflow*:

A workflow whose input data variables are bound to data objects identifiers registered in the data catalog.

Formally, a workflow  $\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$  where

- $\forall dv \in DV_i$  that is an input data variable  $\exists d \in D : \langle dv, d \rangle \in DVB_w$

where  $DV_i$  denotes the set of input data variables in the workflow:

$$DV_i \equiv \{ dv \in DV_w \mid \exists n \in N_w : \exists i \in I : \langle \perp, \perp, dv, n, i \rangle \in L_w \}$$

— *Configured workflow:*

A workflow where all the parameters of its components have been assigned values.

Formally, a workflow  $\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$  where:

- $\forall pv \in PV_w : \exists v \in V : \langle pv, v \rangle \in PVB_w$

— *Ground workflow:*

A workflow where all the data variables in the workflow have been assigned data object identifiers.

Formally, a workflow  $\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$  where:

- $\forall dv \in DV_w : \exists d \in D : \langle dv, d \rangle \in DVB_w$

With these definitions at hand, we can make the following distinction:

— *Workflow instance.*

A specialized bound workflow which is also configured. A ground workflow is a special case of workflow instance where all data variables are bound to data objects with assigned identifiers.

— *Workflow template.*

Any workflow that is not a workflow instance.

The workflow in Figure 4 is not specialized, since both nodes contain abstract components. It is partially bound, since it has a binding for the training data variable but not for the test data variable. It is partially configured, since only some of its parameter variables have been assigned values. This workflow is a workflow template, and we will use it as a running example to show how our algorithms use it to create a workflow instance by specializing, binding, and configuring it. The workflow in Figure 5 is specialized, partially bound, and partially configured.

A workflow library containing reusable workflows can include any kind of workflow template, whether they are fully or partially specialized, bound, or configured. Different kinds of workflow templates can be reused for different purposes. For example, a fully configured workflow can be reused to process different datasets, while a fully bound workflow can be reused to explore alternative parameter settings.

In contrast with workflow templates, workflow instances are fully specified in terms of data, parameters, and components to be used. Therefore, workflow instances can be

submitted to a workflow mapping and execution engine to be mapped to available execution resources and to be subsequently executed.

In order to be submitted to the workflow mapping and execution system, workflow instances need to have unique data object identifiers for each new data product as well as exact command line invocations for each component. The workflow mapping and execution system does not need the DODs and other constraints that may be included in the workflow as a result of its evolution from a workflow template to a workflow instance. It only needs to have a unique identifier for each new dataset that will result from the execution of the workflow, specific mention of codes to be executed for each component, and an invocation command to invoke each component code. We refer to these workflows as *ground workflow instances*, where only the basic structure of the workflow is given and no data variables or metadata are included.

A workflow catalog can support analogous functions to component catalogs:

- `is-configurable(workflow)`
- `configure(workflow)`
- `is-backward-enabled(workflow)`
- `find-DODs-given-output-requirements(workflow)`
- `is-forward-enabled(workflow)`
- `predict-DODs-given-input-requirements(workflow)`
- `estimate-performance(workflow)`

The functions **find-DODs-given-output-requirements** and **predict-DODs-given-input-requirements** can be used to generate requirements on inputs and predictive metadata on outputs respectively at the workflow level, as an alternative to finding requirements component by component as we will explain in detail later. Similarly to components, we refer to the workflows in the workflow catalog as *self-configurable*, *forward-enabled*, and *backward-enabled* when the corresponding functions are supported.

Later on, we will show how these functions can be used during workflow generation.

### Workflow Requests: Formalization

We discussed earlier a broad range of requirements that users could provide to a workflow system. We focus here on particular kinds of requests, namely those where a workflow template is given by the user to provide functional and structural properties of the answer to be found by the system. Together with a workflow template from the library, a seed is specified that further constrains data and parameter variables. A given request may contain several pairs of templates and seeds when the user would like the system to consider several templates as a starting point to find the solution. By specifying a template, the user is providing structural properties as indicated by the relative ordering of the steps in the template. Also through the workflow template, the user can provide functional properties since the template specifies component types to be used as well as constraints on data variables.

Formally, given a workflow template library  $L$ , a component catalog  $C$ , and a data catalog  $D$ , a request  $WR$  is defined as a pair of a workflow template and a seed:

$$\langle w_r, S_r \rangle$$

where  $w_r \in L$  is a workflow template defined by a tuple

$$\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$$

and the seed  $S_r$  is defined by a tuple:

$$\langle DV_r, PV_r, M_r, DVB_r, PVB_r \rangle$$

where:

- $DV_r$  is a set of data variables for the seed. A subset of these variables may be specified to be input variables  $IDV_r \subseteq DV_r$  and another subset may be specified to be output variables  $ODV_r \subseteq DV_r$ .
- $PV_r$  is a set of parameter variables for the seed.
- $M_r$  is a (possibly empty) set of DODs using the variables in  $DV_r$  and in  $PV_r$ .
- $DVB_r$  is a (possibly empty) set of bindings of the workflow data variables to data object identifiers:

$$\langle dv, d \rangle \equiv \langle dv, \text{has-value}, d \rangle$$

$$dv \in DV_r, d \in D$$

- $PVB_r$  is a (possibly empty) set of bindings of the parameter variables to values:

$$\langle pv, v \rangle \equiv \langle pv, \text{has-value}, v \rangle$$

$$pv \in PV_r, v \in V$$

We use the following definitions:

— *Unified request:*

A request where the data and parameter variables in the seed correspond to the data and parameter variables of the workflow template specified in the request.

Formally,  $DV_r \subseteq DV_w$  and  $PV_r \subseteq PV_w$

— *Well-formed request:*

A request where any bindings and values in the seed for data and parameter variables do not overlap with the bindings specified in the workflow template.

Formally:

$$\text{▪ } DVB_r \cap DVB_w = \emptyset$$

$$\text{▪ } PVB_r \cap PVB_w = \emptyset$$

— *Bound request:*

A unified well-formed request where the seed and the workflow template provide bindings for all the input data variables of the workflow template, and the bindings do not overlap.

Formally:

- $DVB_r \cap DVB_w = \emptyset$
- $\forall dv \in DV_w \text{ then } \exists \langle dv, d \rangle \in DVB_r \cup DVB_w$
- *Configured request:*  
A unified well-formed request where the seed and the workflow template provide values for all the parameter variables of the workflow template specified in the request, and the value assignments do not overlap.

**Table 2: Formal Representation of a Request**

---

```

<ModelerThenClassifier, {ClassificationDataVariable}, {ClassIndexParameterVar},
  {<ClassificationDataVariable hasDomain weather>
   <ClassificationDataVariable hasType Classification>}, {}
  {<ClassIndexParameterVar 5>}>

```

---

Formally:

- $PVB_r \cap PVB_w = \emptyset$
- $\forall pv \in PV_w \text{ then } \exists \langle pv, v \rangle \in PVB_r \cup PVB_w$
- *Seedless request:*  
A request where the seed is empty.

Table 2 shows an example request specifying the workflow to be used (ModelerThenClassifier) and providing DODs on a data variable (ClassificationDataVariable) and a binding for a parameter variable (ClassIndexParameterVar).

We define an additional type of workflow:

- *Seeded workflow*, or workflow seeded with a request:  
A workflow where the DODs for the variables in the request have been combined with the DODs of the workflow template specified in the request, and the bindings and parameter values in the request have been combined with those of the workflow template specified in the request. In order to create a seeded workflow, the request has to be unified and well-formed.

Formally, a request with the workflow template:

$$\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$$

and the seed:

$$\langle DV_r, PV_r, M_r, DVB_r, PVB_r \rangle$$

results in the seeded workflow:

$$\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$$

where:

$$DV_r \subseteq DV_w$$

$$DVB_r \cap DVB_w = \emptyset$$

$$\begin{aligned}
PV_r &\subseteq PV_w \\
PVB_r \cap PVB_w &= \emptyset \\
M_n &= \text{d:combine-DODs}(M_r, M_w) \\
DVB_n &= DVB_r \cup DVB_w \\
PVB_n &= PVB_r \cup PVB_w
\end{aligned}$$

Typically a request will contain a single template and a single seed. We generalize this by allowing a request to contain several template and seed pairs. By specifying several templates and seeds, the user would intend to provide the system with a broader pool of candidate workflows to search through in generating a solution.

If a workflow template were not specified in the request and only the seed was, the system would have to retrieve relevant templates from the workflow library, which would often result on several matching templates. Therefore, our algorithm would be preceeded by a step to retrieve matching templates for the seed. After the templates were retrieved, we would have a workflow request of the form considered here and our algorithms would take on the remainder of the process.

The next section describes the algorithms for generating workflows from these workflow requests.

### 3.1.3 Algorithm: Automatic Template-Based Workflow Generation

The workflow generation process starts with a request containing several template/seed pairs. We assume we start with a unified and well-formed request, that is, the variables that appear in the seed are a subset of the variables in the workflow template and any bindings specified in the seed do not overlap with the bindings specified in the workflow template.

Throughout this section, we use the following conventions. The variables of the algorithm are shown in *italics*. The functions shown in all capital letters are elaborated in later subsections. We assume some functions have been defined with the prefixes “get-“ or “set-“ on workflow and request data structures to access their individual constituents. The function calls in **boldface** are functions supported by external catalogs, using **c:** as a prefix for function calls to external component services, **d:** for metadata services, and **w:** for workflow services.

Since the algorithms perform function calls to external services, it includes provisions for function calls returning a special error code (the empty set) when there is either some error in the inputs to the function call or the function is undefined for those inputs. In such cases, the algorithms reject the workflow being considered as a candidate. These may be indications that the external catalogs may need to be extended to refine their models or to include additional components or data objects.

#### Top-Level Algorithm

Figure 3 shows an overview of the distinct stages during workflow generation. Table 3 describes the top-level algorithm for automatic template-based workflow generation. The algorithm analyzes workflow candidates at each level on a breadth-first manner, that is,

all candidates are elaborated before proceeding to elaborate workflows at the next level of detail. A depth-first search version of the algorithm is also a possible alternative. In either case, the approach we take is to generate all possible candidates, and then rank them to select the top choices. This is needed because ranking candidates properly requires that the workflow is specialized and configured.

The algorithm begins by creating a seeded workflow from each of the template/seed pair in the request. If there were any errors seeding the workflow, due to inconsistencies in the definition of the seed and the template, the call to SEED-WORKFLOW would return an empty set and the whole procedure would be terminated.

Next, the algorithm elaborates the workflows and in the process it will find the requirements on input data. For each candidate workflow, it will start from the output links and retrieve any additional constraints on the workflow variables that are required in order to produce the required output. We refer to this process as a *backward sweep*. This is done within the algorithm BACKWARD-SWEEP, which we describe in detail below. When the workflows contain abstract components, the backward sweep algorithm may find more specific component classes that are appropriate to satisfy the requirements. When this occurs, several candidate workflows are returned. At the end of the backward sweep, the original DODs of the workflows have been augmented and can be used to find datasets that match the request and workflow requirements. We refer to these as *binding-ready workflows*.

Now that there are as many constraints on the input data as could be uncovered in the backward sweep, the algorithm retrieves appropriate input data sources. This is done by the algorithm SELECT-INPUT-DATA-OBJECTS, which essentially generates bindings for all the input data variables of the workflow that are not bound in the original request. Because there may be several alternative datasets that are appropriate, several alternative bindings may be found and in that case several candidate workflows are returned. For each workflow candidate, all the properties of the input data sources that may be relevant to the request are incorporated into the workflow. At the end of this process, the workflow candidates are all bound.

Next, the algorithm elaborates the workflows by propagating the properties of input data sources through each step of the workflow. Starting from the input links, it will retrieve any additional constraints on workflow variables that result from the properties of input data sources. This process is called a *forward sweep*. It is done with the algorithm FORWARD-SWEEP, which is described below. These workflows have augmented DODs that result from propagating the properties of the input data, and we refer to them as *elaborated workflows*. The workflow may still contain abstract components, and the forward sweep algorithm would specialize them. This results in several candidate workflows being returned. The forward sweep also assigns values to all the parameters of the workflow components based on the constraints that are known for the workflow variables at each step. At the end of this process, the workflow candidates are all specialized and configured in addition to being elaborated.

Next, all the candidate workflows are ranked based on estimates of their performance. This ranking function only takes into account a rough estimate of the execution time of a component based on characteristics of the data. It does not take into account the different performance across architectures or other characteristics of the execution host such as

memory availability. These estimates also do not take into account how the workflow performance is affected by data movements, queue wait times, and other execution delays. Such finer-grained estimates are produced by the workflow mapping and execution system and are not discussed here. The rough estimates used at this stage are generated by the algorithm ESTIMATE-PERFORMANCE, described below. The k-best workflows are returned.

**Table 3: Top-Level Algorithm for Automatic Template-Based Workflow Generation**

Algorithm: **TEMPLATE-BASED-WORKFLOW-GENERATION**

Input: *request*

Output: *workflow-instances*

```

Seed Workflows from Request
1   seeded-workflows  $\leftarrow \{\}$ 
2   for each template-seed-pair  $\in$  request do
3       workflows  $\leftarrow$  SEED-WORKFLOW(template-seed-pair)
4       if (workflows  $\neq$  null) then
5           seeded-workflows  $\leftarrow$  seeded-workflows  $\cup$  workflows
6       if (seeded-workflows = null) then workflow-instances  $\leftarrow \{\}$ ; return

Find Input Data Requirements
7   binding-ready-workflows  $\leftarrow \{\}$ 
8   for each seeded-workflow  $\in$  seeded-workflows do
9       workflows  $\leftarrow$  BACKWARD-SWEEP(seeded-workflow)
10      if (workflows  $\neq$  null) then
11          binding-ready-workflows  $\leftarrow$  binding-ready-workflows  $\cup$  workflows
12      if (binding-ready-workflows = null) then workflow-instances  $\leftarrow \{\}$ ; return

Data Source Selection
13  bound-workflows  $\leftarrow \{\}$ 
14  for each binding-ready-workflow  $\in$  binding-ready-workflows do
15      workflows  $\leftarrow$  SELECT-INPUT-DATA-OBJECTS(binding-ready-workflow)
16      if (workflows  $\neq$  null) then
17          bound-workflows  $\leftarrow$  bound-workflows  $\cup$  workflows
18      if (bound-workflows = null) then workflow-instances  $\leftarrow \{\}$ ; return

Parameter Selection
19  configured-workflows  $\leftarrow \{\}$ 
20  for each bound-workflow  $\in$  bound-workflows do
21      workflows  $\leftarrow$  FORWARD-SWEEP(bound-workflow)
22      if (workflows  $\neq$  null) then
23          configured-workflows  $\leftarrow$  configured-workflows  $\cup$  workflows
24      if (configured-workflows = null) then workflow-instances  $\leftarrow \{\}$ ; return

Workflow Ranking
25  ranked-workflows  $\leftarrow \{\}$ 
26  for each configured-workflow  $\in$  configured-workflows do
27      workflow  $\leftarrow$  ESTIMATE-PERFORMANCE(configured-workflow)

```

---

```

28         ranked-workflows  $\leftarrow$  ranked-workflows  $\cup$  {workflow}
29         ranked-workflows  $\leftarrow$  select-k-best(ranked-workflows)

Workflow Instatiation and Grounding
30         workflow-instances  $\leftarrow$  {} ground-workflows  $\leftarrow$  {}
31         for each ranked-workflow  $\in$  ranked-workflows do
32             workflow  $\leftarrow$  INSTANTIATE-WORKFLOW(ranked-workflow)
33             workflow-instances  $\leftarrow$  workflow-instances  $\cup$  {workflow}
34             ground-workflows  $\leftarrow$  GROUND-WORKFLOW(workflow)
35         return ground-workflows

```

---

**Table 4: Algorithm for Seeding a Workflow Template with the Seed Given in the Request**

---

Algorithm: **SEED-WORKFLOW**

Input: *template-seed-pair*  
Output: *seeded-workflow*

```

1         workflow  $\leftarrow$  get-template(template-seed-pair)
2         seed  $\leftarrow$  get-seed(template-seed-pair)
        Combine DODs of the seed with the workflow DODs
3         workflow-DODs  $\leftarrow$  get-DODs(workflow)
4         seed-DODs  $\leftarrow$  get-DODs(seed)
5         new-DODs  $\leftarrow$  d:combine-DODs(workflow-DODs, seed-DODs)
        If the DODs are inconsistent, reject the current workflow
6         if new-DODs = {} then
7             workflow  $\leftarrow$  {}
8         else
9             set-DODs(workflow, new-DODs)
        Combine the data variable bindings of the seed with the workflow data variable bindings
10        workflow-data-var-bindings  $\leftarrow$  get-data-var-bindings(workflow)
11        seed-data-var-bindings  $\leftarrow$  get-data-var-bindings(seed)
12        new-data-var-bindings  $\leftarrow$  workflow-data-var-bindings  $\cup$  seed-data-var-bindings
13        set-data-var-bindings(workflow, new-data-var-bindings)
        Combine the parameter bindings of the seed with the workflow parameter bindings
14        workflow-par-var-bindings  $\leftarrow$  get-par-var-bindings(workflow)
15        seed-par-var-bindings  $\leftarrow$  get-par-var-bindings(seed)
16        new-par-var-bindings  $\leftarrow$  workflow-par-var-bindings  $\cup$  seed-par-var-bindings
17        set-par-var-bindings(workflow, new-par-var-bindings)
18        return workflow

```

---

The algorithm then proceeds to ground the selected workflows by assigning unique logical identifiers to variables in the workflow that are not input data variables nor parameter variables. For each intermediate and final link in the workflow, its corresponding variable will be assigned a unique identifier using the DODs that describe

its properties. During this step the invocation command for each component is also formulated. This is done by the algorithm INSTANTIATE-WORKFLOW described below. All workflow candidates will then be ground and ready to be formatted for submission to the workflow mapping and execution system by extracting from the workflow instance only the information required for a ground workflow.

**Table 5: Algorithm for Backward Sweep Through Workflow**

---

Algorithm: **BACKWARD-SWEEP-THROUGH-WORKFLOW**

Input: *seeded-workflow*

Output: *binding-ready-workflows*

```

1   workflow ← seeded-workflow
2   new-DODs ← w:find-DODs-given-output-requirements(workflow)
3   when (new-DODs ≠ {})
4       set-DODs(workflow, get-DODs(workflow) ∪ new-DODs)
5   binding-ready-workflows ← {workflow}
6   return binding-ready-workflows

```

---

### Generating Seeded Workflows

First, the DODs of the seed and the DODs of the workflow template of the request are combined. If the DODs of the seed and the workflow are inconsistent, the call to the metadata services to combine these DODs will indicate an error by returning an empty set. In that case, an empty seeded workflow is returned to the top-level algorithm. Next, the data variable bindings of the seed and the workflow template are combined. Finally, the parameter bindings of the seed and the workflow are combined. Since we assume that the request is unified and well-formed, no errors will occur when merging the bindings. The result of this stage is a seeded workflow.

### Backward Sweep

The backward sweep can obtain the constraints on the input data variables in two different ways. One way is to use workflow services. These services would propagate the constraints at the workflow level and would not necessarily reason about constraints for the intermediate variables in the workflow. Another way is to use component services. The algorithm would have to walk through the workflow nodes and propagate constraints component by component by invoking functions implemented by the component catalog for each of the components.

Table 5 shows the algorithm for the BACKWARD-SWEEP function using the workflow services. A single function that takes the whole workflow as an argument will return any additional DODs including DODs on input data variables but may also contain DODs on intermediate data variables when appropriate.

**Table 6: Algorithm for Backward Sweep through Components**

Algorithm: **BACKWARD-SWEEP-THROUGH-COMPONENTS**

Input: *seeded-workflow*

Output: *binding-ready-workflows*

```

1    workflow-queue ← seeded-workflow
2    binding-ready-workflows ← {}
3    while (workflow-queue ≠ {}) do
4        workflow ← dequeue(workflow-queue)
7        link-queue ← get-output-links(workflow)
8        while (link-queue ≠ {} & workflow ≠ {}) do
9            link ← dequeue(link-queue)
10           when (current-link ∉ get-input-links(workflow))
11               node ← get-origin(link)
12               Find all links (going sideways) that have the current node as the origin node
13               links-shared-origin ← l s.t. l ∈ get-links(workflow) & get-origin(l) = node
14               link-queue ← link-queue \ links-shared-origin
15               Find all links (going upstream) that have the current node as the destination node
16               links-shared-dest ← l s.t. l ∈ get-links(workflow) & get-destin(l) = node
17               link-queue ← link-queue \ links-shared-dest
18               Create a set with all those links that have the current node as origin or destination
19               links-current-node ← links-same-origin ∪ links-dest-is-origin
20               Find all the DODs in the workflow that are relevant to the current node
21               vars ← get-data-vars(workflow) ∪ get-param-vars(workflow)
22               vars-node ← v s.t. v ∈ vars & l ∈ links-current-node & get-variable(l) = v
23               node-DODs ← entity-DODs(vars-node, get-DODs(workflow))
24               comp ← get-node-component(node)
25               Map DODs on workflow variables to DODs on arguments of the node's component
26               comp-DODs ← find-comp-DODs(node, comp, node-DODs, links-current-node)
27               If the node's component is not concrete, get specializations of the component
28               and create a new workflow candidate with each of the specializations obtained
29               if (not c:is-concrete(comp)) then
30                   concrete-components ← c:specialize(comp, comp-DODs)
31               If no specialization of the component can satisfy the requirements, reject the current workflow
32               when (concrete-components ≠ {})
33                   for each cc ∈ concrete-components do
34                       copy ← copy(workflow)
35                       copy ← replace(comp, cc, node, copy)
36                       workflow-queue ← workflow-queue ∪ copy
37               else
38                   comp-input-DODs ← c:find-DODs-given-output-requirements(comp, comp-DODs)
39                   If no DODs can satisfy the requirements on the component, reject the current workflow
40                   if comp-input-DODs = {}
41                       workflow ← {}
42                   else
43                       Map DODs on arguments of the node's component to DODs on workflow variables
44                       var-DODs ← find-variable-DODs(vars, node, comp, comp-input-DODs)
45                       set-DODs(current-workflow) ← get-DODs(workflow) ∪ var-DODs
46               end while over link-queue
47           when (workflow ≠ {})
48               binding-ready-workflows ← binding-ready-workflows ∪ workflow
49       end while over workflow-queue
50   return binding-ready-workflows

```

**Table 7: Algorithm for Binding Workflows by Selecting Input Data**

---

Algorithm: **SELECT-INPUT-DATA-OBJECTS**

Input: *binding-ready-workflow*

Output: *bound-workflows*

```

1   bound-workflows  $\leftarrow \{\}$ 
2   input-links  $\leftarrow$  get-input-links(specialized-workflow)
3   input-data-variables  $\leftarrow$  get-variables(input-links)
4   input-DODs  $\leftarrow$  get-variables(input-data-variables)
5   input-bindings  $\leftarrow$  d:find-data-objects(input-DODs)
6   for each binding  $\in$  input-bindings do
7       workflow  $\leftarrow$  copy(specialized-workflow)
8       set-data-variable-bindings(workflow, get-data-variable-bindings(workflow)  $\cup$  binding)
9       data-objects  $\leftarrow$  get-data-objects(input-bindings)
10      for each data-object  $\in$  data-objects do
11          additional-DODs  $\leftarrow$  d:obtain-DODs(data-object)
12          set-DODs(workflow, get-DODs(workflow)  $\cup$  additional-DODs)
13      bound-workflows  $\leftarrow$  bound-workflows  $\cup$  {workflow}
14  return bound-workflows

```

---

**Table 8: Algorithm for Forward Sweep Through Workflow**

---

Algorithm: **FORWARD-SWEEP-THROUGH-WORKFLOW**

Input: *bound-workflow*

Output: *configured-workflow*

```

1   workflow  $\leftarrow$  bound-workflow
2   new-DODs  $\leftarrow$  w:predict-DODs-given-input-requirements(workflow)
3   when (new-DODs  $\neq \{\}$ )
4       set-DODs(workflow, get-DODs(workflow)  $\cup$  new-DODs)
5   if is-configured(workflow)
6       configured-workflow  $\leftarrow$  workflow
7   else
8       configured-workflow  $\leftarrow$  null
9   return configured-workflow

```

---

Table 6 shows the algorithm for the BACKWARD-SWEEP function using the component services. For each node in the candidate workflow, it traverses the workflow from end results to initial inputs. For each of the nodes visited, the algorithm processes together all the links that have that node either as an origin or as a destination. This is because some workflow nodes are origin to more than one link. In such cases, we need

to gather all the DODs on workflow variables that constrain the parameters of that node's component as we traverse the workflow. If the component is abstract, all the possible specializations from that abstract component class are obtained. These could be either more specific component classes or concrete components. When more than one specialization is returned, more than one specialized workflow will be created for the initial seeded workflow. All additional DODs that the component places on its arguments and that are returned by the component catalog are added to the workflow DODs. This may include parameter values that can be set during this step as constraints on input and output arguments of the component are introduced by the workflow.

For simplicity, the algorithm in Table 7 assumes that each node in the workflow is origin to only one link. In cases where there is more than one link with the node as origin, the algorithm will only proceed to specialize the component in a node when all links relevant to the outputs of a node have been processed. That is, it ensures that the paths from the outputs to that node have already been fully processed.

When using the workflow services for the backward sweep, any abstract components of the workflow will not be specialized. Therefore, when using workflow services for the backward sweep the workflow template specified in the request must be a concrete workflow.

The result of the backward sweep is a set of candidate workflows that are all binding-ready workflows.

### Selecting Input Data

This algorithm starts with a binding-ready workflow that includes DODs on all input data variables. First, it finds available data objects that match those DODs. There can be several combinations of data object for input data variables, and in that case several sets of bindings are returned. In that case, a bound workflow will be created for each set of bindings. If there are no matching data sources then the workflow is rejected and an empty workflow is returned.

Note that there is a single query to the data catalog for a given workflow, rather than a query per input data variable. This ensures that any constraints among input data variables are taken into account by the data catalog during the matching of input data sources.

Next, the algorithm requests from the metadata services all additional DODs of the selected input data objects. There may be arbitrarily many possible properties of a data object and there may be a cost to generating the values of some of the properties. Ideally, this function would be invoked in a selective and cost-sensitive manner though this is not addressed in our current work.

### Forward Sweep

Like the backward sweep, the forward sweep can be approached in two different ways. One approach is to use workflow services. These services would propagate the constraints on input data variables at the workflow level and would not necessarily reason about constraints for the intermediate variables in the workflow. Another approach is to

use component services. The algorithm would have to walk through the workflow nodes and propagate constraints component by component by invoking functions implemented by the component catalog for each of the components.

Table 9: Algorithm for Forward Sweep Through Components

Algorithm: **FORWARD-SWEEP-THROUGH-COMPONENTS**

Input: *bound-workflow*

Output: *configured-workflows*

```

1   workflow-queue  $\leftarrow$  bound-workflow
1   configured-workflows  $\leftarrow$  {}
3   while (workflow-queue  $\neq$  {}) do
4       workflow  $\leftarrow$  dequeue(workflow-queue)
2   link-queue  $\leftarrow$  get-input-links(workflow)
8   while (link-queue  $\neq$  {} & workflow  $\neq$  {}) do
9       link  $\leftarrow$  dequeue(link-queue)
10      when (current-link  $\notin$  get-output-links(workflow))
11          node  $\leftarrow$  get-dest(link)
      Find all links (going sideways) that have the current node as the origin node
12          links-shared-origin  $\leftarrow$   $l$  s.t.  $l \in$  get-links(workflow) & get-origin( $l$ ) = node
13          link-queue  $\leftarrow$  link-queue  $\setminus$  links-shared-origin
      Find all links (going upstream) that have the current node as the destination node
14          links-shared-dest  $\leftarrow$   $l$  s.t.  $l \in$  get-links(workflow) & get-destin( $l$ ) = node
15          link-queue  $\leftarrow$  link-queue  $\setminus$  links-shared-dest
16          links-current-node  $\leftarrow$  links-same-origin  $\cup$  links-dest-is-origin
      Find all the DODs in the workflow that are relevant to the current node
17          vars  $\leftarrow$  get-data-vars(workflow)  $\cup$  get-param-vars(workflow)
18          vars-node  $\leftarrow$   $v$  s.t.  $v \in$  vars &  $l \in$  links-current-node & get-variable( $l$ ) =  $v$ 
19          node-DODs  $\leftarrow$  d:entity-DODs(vars-node, get-DODs(workflow))
20          comp  $\leftarrow$  get-node-component(node)
      Map DODs on workflow variables to DODs on arguments of the node's component
21          comp-DODs  $\leftarrow$  find-comp-DODs(node, comp, node-DODs, links-current-node)
      If the node's component is not concrete, create a new workflow candidate with each specialization
22          if (not c:is-concrete(comp)) then
23              concrete-components  $\leftarrow$  c:specialize-to-concrete(comp, node-DODs)
      If no specialization of the component can satisfy the requirements, reject the current workflow
24          when (concrete-components  $\neq$  {})
25              for each cc  $\in$  concrete-components do
26                  copy  $\leftarrow$  copy(workflow)
27                  copy  $\leftarrow$  replace(comp, cc, node, copy)
28                  workflow-queue  $\leftarrow$  workflow-queue  $\cup$  copy
29          else
      If the component is not configured, create a new workflow candidate with each configuration obtained
30          if (not c:is-configured(comp, comp-DODs)) then
31              component-configurations  $\leftarrow$  c:configure(comp, comp-DODs)
      If no configuration of the component can satisfy the requirements, reject the current workflow
      when (component-configurations  $\neq$  {})
32          for each cc  $\in$  component-configurations do
33              new  $\leftarrow$  copy(workflow)
34              new  $\leftarrow$  replace-component(workflow, cc)
35              workflow-queue  $\leftarrow$  workflow-queue  $\cup$  copy
36          else
37              comp-o-DODs  $\leftarrow$  c:predict-DODs-given-input-requirements(comp, comp-DODs)
      Map DODs on arguments of the node's component to DODs on workflow variables
38          var-DODs  $\leftarrow$  find-variable-DODs(vars, node, comp, comp-o-DODs)
39          set-DODs(current-workflow, get-DODs(workflow)  $\cup$  var-DODs)
40      end while over link-queue
41  when (workflow  $\neq$  {})

```

```

42         configured-workflows  $\leftarrow$  configured-workflows  $\cup$  {workflow}
43     end while over workflow-queue
44     return configured-workflows

```

**Table 10: Algorithm for Estimating Workflow Performance Through the Workflow**

---

Algorithm: **ESTIMATE-PERFORMANCE-THROUGH-WORKFLOW**

Input: *ground-workflow*

Output: *ranked-ground-workflow*

```

1     workflow  $\leftarrow$  ground-workflow
2     set-performance-estimate(workflow, w: estimate-performance(workflow) )
3     return workflow

```

---

**Table 11: Algorithm for Estimating Workflow Performance Through Components**

---

Algorithm: **ESTIMATE-PERFORMANCE-THROUGH-COMPONENTS**

Input: *ground-workflow*

Output: *ranked-ground-workflow*

```

1     workflow  $\leftarrow$  ground-workflow
2     link-queue  $\leftarrow$  get-output-links(workflow)
3     while (link-queue  $\neq$  { }) do
4         link  $\leftarrow$  dequeue(link-queue)
5         when (current-link  $\notin$  get-output-links(workflow))
6             node  $\leftarrow$  get-dest(link)
            Find all links (going sideways) that have the current node as the origin node
7             links-shared-origin  $\leftarrow$  l s.t. l  $\in$  get-links(workflow) & get-origin(l) = node
8             link-queue  $\leftarrow$  link-queue  $\setminus$  links-shared-origin
            Find all links (going upstream) that have the current node as the destination node
9             links-shared-dest  $\leftarrow$  l s.t. l  $\in$  get-links(workflow) & get-destin(l) = node
10            link-queue  $\leftarrow$  link-queue  $\setminus$  links-shared-dest
11            links-current-node  $\leftarrow$  links-same-origin  $\cup$  links-dest-is-origin
            Find all the DODs in the workflow that are relevant to the current node
12            vars  $\leftarrow$  get-data-vars(workflow)  $\cup$  get-param-vars(workflow)
13            vars-node  $\leftarrow$  v s.t. v  $\in$  vars & l  $\in$  links-current-node & get-variable(l) = v
14            node-DODs  $\leftarrow$  d:entity-DODs(vars-node, get-DODs(workflow))
15            comp  $\leftarrow$  get-node-component(node)
            Map DODs on workflow variables to DODs on arguments of the node's component
16            comp-DODs  $\leftarrow$  find-comp-DODs(node,comp,node-DODs,links-current-node)
            Get estimate of the component performance
17            set-predicted-execution-time(node, c:estimate-performance(comp,comp-DODs) )
18        end while over link-queue
19        set-performance-estimate(workflow, estimate-aggregate-performance(workflow) )
20    return workflow

```

---

As was the case with the backward sweep, the algorithm that uses the workflow services for the backward sweep does not specialize components. Therefore, when using workflow services for the forward sweep the workflow must be a concrete workflow.

The result of the forward sweep is a set of candidate workflows that are all configured and specialized workflows.

Table 12: Algorithm for Instantiating Workflows

---

Algorithm: **INSTANTIATE-WORKFLOW**

Input: *configured-workflow*

Output: *workflow-instances*

```

1    workflow-queue  $\leftarrow$  configured-workflow
2    workflow-instances  $\leftarrow$  {}
3    while (workflow-queue  $\neq$  {}) do
4        workflow  $\leftarrow$  dequeue(workflow-queue)
5        link-queue  $\leftarrow$  get-output-links(workflow)
6        while (link-queue  $\neq$  {} & workflow  $\neq$  {}) do
7            link  $\leftarrow$  dequeue(link-queue)
8            when (current-link  $\notin$  get-input-links(workflow))
9                link-DODs  $\leftarrow$  d:entity-DODs(get-variable(current-link), get-DODs(workflow))
10               id  $\leftarrow$  d:assign-identifier(link-DODs)
11               binding  $\leftarrow$  <get-link-variable(link), id>
12               d:assert-predicted-DODs(id, link-DODs)
13               set-workflow-bindings(workflow, get-workflow-bindings(workflow)  $\cup$  binding )
14         end while over link-queue
14         set-invocation-commands(workflow)
15     workflow-instances  $\leftarrow$  workflow-instances  $\cup$  {workflow}
16     end while over workflow-queue
17     return workflow-instances

```

---

### Estimating Workflow Performance

Like the forward and backward sweeps, estimating workflow performance can be done using workflow services or component services. Because the estimates using component services would need the DODs for intermediate data products, it is required that the forward sweep should have been done using component services as well.

The algorithm to estimate performance using component services is shown in Table 11. It walks through the nodes of the workflow, and for each node it gathers the DODs that are relevant to it. Using those DODs, it invokes the component services to retrieve the estimates of performance of the workflow. With the individual estimates for each node, the algorithm then calls a function that aggregates the estimates for the overall workflow (**estimate-aggregate-performance**). This function finds the longest path between the input links and the output links.

### Instantiating and Grounding Workflows

The algorithm for instantiating workflows, shown in Table 12, traverses a workflow and gathers all the DODs on a data variable and requests from the data catalog a unique identifier for the corresponding execution data product. If the DODs are rich enough, the data catalog will be able to detect when data products are equal and therefore give them the same identifier. This enables data reuse with the benefit of saving computation time, as the workflow execution system can eliminate unnecessary computations that produce already existing data that was produced by previously executed workflows. When the DODs are not rich enough, then reuse will not be possible as each new data product will have its own identifier and there will be no way to detect when data products from different workflows are the same (unless the workflows are identical). This can happen in the case where the forward sweep proceeds through workflows rather than through components.

During grounding of workflows, the invocation command is set for each of the node's components. A function is shown in the table that applies to the whole workflow, within that function there is an invocation of the component catalog for each node's component using all the DODs that are relevant to the variables in links adjacent to the node.

The final grounding step essentially extracts a small subset of the information in the workflow instance to create a ground workflow that can be submitted to the workflow mapping and execution engine. An example of a workflow instance and its corresponding ground workflow is shown in the next section.

### Summary of Functions for Data and Catalog Services

Table 13 summarizes the functions for data and catalog services invoked by the workflow generation algorithm. For each function, we indicate the use of that function in the algorithm.

The next section walks through the main steps of the algorithm with an example of a workflow request.

#### **3.1.4 A walkthrough Example of Workflow Generation**

We now show an example of how workflow candidates are generated from a workflow request using the algorithm just presented. We present an example that runs end-to-end in our implemented system, and show through the representation of candidate workflows at each stage. We use different namespaces to refer to terms that are defined in different catalogs. Therefore, the workflow catalog, component catalog, and data catalog will have different namespaces. We use the W3C Ontology Web Language OWL ([www.w3.org/TR/owl-features](http://www.w3.org/TR/owl-features)) to represent workflows and DODs, but will show the examples using N3 notation.

Table 14 shows the representation of the workflow template for `ModelerThenClassifier` shown in Figure 4. The workflow contains two nodes for a modeler and a classifier. There are six links that represent inputs and outputs of the two nodes. Note that the workflow in Figure 4 shows the heap size as a parameter of the classifier, which is not used in this example.

Table 15 shows an example of a representation of a workflow request. It specifies the workflow to use (ModelerThenClassifier) and provides additional DODs on an output data variable and a parameter variable. In particular, the output of the workflow should include a classification of a weather data object (i.e. the domain of the ClassificationDataVariable is Weather) and the value of the ClassIndexParameterVar is 5. This is the same request shown in Table 2.

**Table 13: Summary of functions that need to be supported in the metadata services and the component services to enable automatic workflow generation**

| Function in Metadata Services | Purpose in Automatic Generation Process   |
|-------------------------------|---|
| d:combine-DODs                | Seed workflow templates   |
| d:entity-DODs                 | Filter relevant data properties to be propagated in the workflow enables workflow candidate addition and elimination      |
| d:assign-identifier           | Create unique identifiers and properties for workflow data products so they can be reused in future workflows             |
| d:assert-predicted-DODs       |   |
| d:find-data-objects           | Selection of input data enables creation of bound workflows   |
| d:obtain-DODs                 | Propagation of input data properties in forward sweep enables component specialization and workflow candidate elimination |

| Function in Component Services          | Purpose in Automatic Generation Process  |
|---|--|
| c:inputs                                | Validate workflow template in request  |
| c:parameters                            |  |
| c:outputs                               |  |
| c:args                                  |  |
| c:invocation-command                    | Ground workflows to be submitted for execution   |
| c:is-concrete                           | Use of abstract components in workflow templates that can be specialized in backward and forward sweep |
| c:specialize                            |  |
| c:specialize-to-concrete                |  |
| c:is-backward-enabled                   | Generate binding-ready workflows in backward sweep   |
| c:find-DODs-given-output-requirements   | Generate configured workflows in forward sweep   |
| c:is-forward-enabled                    |  |
| c:predict-DODs-given-input-requirements |  |
| c:is-configurable                       |  |
| c:configure                             |  |
| c:is-configured                         | Rank candidate workflows   |
| c:estimate-performance                  |  |

Given this request, the seeded workflow initially generated by the algorithm is as the original template shown in Table 13 except that it includes the additional DODs on `ClassificationDataVariable` and `ClassIndexParameterVar` that are introduced by the request. Table 16 shows a relevant excerpt of the seeded workflow, where the additions to the original workflow template are highlighted in bold face.

**Table 14: A Workflow Example in N3 Notation**

---

```
ModelerThenClassifier    a    wflow:WorkflowTemplate ;
    wflow:hasNode classifierNode , modelerNode ;
    wflow:hasLink modelerTrainingDataInputLink, modelerJavaMaxHeapInputLink,
        classifierOutputLink , modelerOutputClassifierInputInOutLink,
        modelerClassIndexInputLink , classifierDataInputLink .

modelerNode      a    wflow:Node ;    wflow:hasComponent ac:Modeler.
classifierNode    a    wflow:Node ;    wflow:hasComponent ac:Classifier .

modelerTrainingDataInputLink    a    wflow:InputLink ;
    wflow:hasDestinationNode modelerNode ;
    wflow:hasDestinationArgument ac:d ;
    wflow:hasVariable TrainingDataVariable .

TrainingDataVariable    a    dcdm:Instance , wflow:DataVariable .

maxJavaHeapSizeModelerParameterVar    a    wflow:ParameterVariable .

modelerJavaMaxHeapInputLink    a    wflow:InputLink ;
    wflow:hasDestinationNode modelerNode ;
    wflow:hasDestinationArgument ac:j ;
    wflow:hasVariable maxJavaHeapSizeModelerParameterVar .

ClassIndexParameterVar    a    wflow:ParameterVariable .

modelerClassIndexInputLink    a    wflow:InputLink ;
    wflow:hasDestinationNode modelerNode ;
    wflow:hasDestinationArgument ac:i ;

ModelDataVariable    a    dcdm:Model , wflow:DataVariable .

modelerOutputClassifierInputInOutLink    a    wflow:InOutLink ;
    wflow:hasDestinationNode classifierNode ;
    wflow:hasDestinationArgument ac:m ;
    wflow:hasOriginNode modelerNode ;
    wflow:hasOriginArgument ac:o ;
    wflow:hasVariable ModelDataVariable .

classifierDataInputLink    a    wflow:InputLink ;
    wflow:hasDestinationNode classifierNode ;
    wflow:hasDestinationArgument ac:d ;
    wflow:hasVariable TestDataVariable .

TestDataVariable    a    dcdm:Instance , wflow:DataVariable ;
    dcdm:notSameObject TrainingDataVariable;
    wflow:hasVariable modelerClassIndex .

classifierOutputLink    a    wflow:OutputLink ;
    wflow:hasOriginNode classifierNode ;
    wflow:hasOriginArgument ac:o ;
    wflow:hasVariable ClassificationDataVariable .

ClassificationDataVariable    a    wflow:DataVariable , dcdm:Classification .
```

---

**Table 15: Example of Workflow Requests in N3 Notation**

---

|  |
|--|
| owl:imports ModelerThenClassifier.owl // use ModelerThenClassifier workflow        |
| ClassificationDataVariable a dcdm:Classification ;<br>dcdm:hasDomain dcdm:weather. |
| ClassIndexParameterVar wflow:hasParameterValue 5.                                  |

---

**Table 16: Relevant Excerpts of a Seeded Workflow**

---

|   |
|---|
| ...   |
| ClassIndexParameterVar a wflow:ParameterVariable ;<br><b>wflow:hasParameterValue 5.</b> // from the request                       |
| ...   |
| ClassificationDataVariable a wflow:DataVariable , dcdm:Classification;<br><b>dcdm:hasDomain dcdm:weather.</b> // from the request |
| ...   |

---

Table 17 shows an example of a binding-ready workflow generated as a candidate after the backward sweep. The original Modeler abstract component has been specialized into LmtModeler and the Classifier into J48Classifier. This specialization introduces some new DODs of the data objects used or created by the components, such as dcdm:hasModelType. The DODs in the original request are propagated by the backward sweep and result in additional DODs on some of the workflow data variables. For example, TrainingDataVariable, ModelDataVariable, and TestDataVariable have a new DOD with a requirement in their domain property that it be weather.

Decision Tree classifiers can use Decision Tree models only and Bayes classifiers can only use Bayes models. Assuming a component catalog that includes three Decision Tree modelers (J48Modeler, LmtModeler, ID3Modeler), three Decision Tree classifiers(J48Classifier, LmtClassifier, ID3Classifier), three Bayes modelers(BayeNetModeler, NaiveBayesModeler, HBNModeler) and three Bayes classifiers (BayeNetClassifier, NaiveBayesClassifier, HBNClassifier and six classifiers), 18 total seeded specialized workflows would be generated as candidates.

The next step of the algorithm finds available data objects for the input data variables. With the workflow in Table 17, the following query for selecting input data objects for two input data variables is generated:

**Table 17: An Example Binding-Ready Workflow After the Backward Sweep**

```
TrainingDataVariable    a    dcdm:Instance , wflow:DataVariable;
                        dcdm:hasDomain dcdm:weather.

TestDataVariable        a    dcdm:Instance , wflow:DataVariable;
                        dcdm:notSameObject TrainingDataVariable;
                        dcdm:hasDomain dcdm:weather.

LmtModelerThenJ48Classifier  a    wflow:WorkflowTemplate ;
                        wflow:hasNode classifierNode , modelerNode ;
                        wflow:hasLink modelerTrainingDataInputLink , modelerJavaMaxHeapInputLink , classifierOutputLink ,
                        modelerOutputClassifierInputInOutLink , modelerClassIndexInputLink , classifierDataInputLink.

modelerNode             a    wflow:Node ;    wflow:hasComponent ac:LmtModeler.
classifierNode           a    wflow:Node ;    wflow:hasComponent ac:J48Classifier.

modelerTrainingDataInputLink  a    wflow:InputLink ;
                        wflow:hasDestinationNode modelerNode ;
                        wflow:hasDestinationArgument ac:d ;
                        wflow:hasVariable TrainingDataVariable ;

TrainingDataVariable
a    dcdm:Instance , wflow:DataVariable;
dcdm:hasDomain dcdm:weather.

maxJavaHeapSizeModelerParameterVar  a    wflow:ParameterVariable .

modelerJavaMaxHeapInputLink  a    wflow:InputLink ;
                        wflow:hasDestinationNode      modelerNode ;
                        wflow:hasDestinationArgument ac:j ;
                        wflow:hasVariable j maxJavaHeapSizeModelerParameterVar .

ClassIndexParameterVar  a    wflow:ParameterVariable .
                        wflow:hasParameterValue 5. // from the request

modelerClassIndexInputLink  a    wflow:InputLink ;
                        wflow:hasDestinationNode      modelerNode ;
                        wflow:hasDestinationArgument ac:i ;
                        wflow:hasVariable ClassIndexParameterVar .

modelerOutputClassifierInputInOutLink  a    wflow:InOutLink ;
                        wflow:hasDestinationNode      classifierNode ;
                        wflow:hasDestinationArgument ac:m ;
                        wflow:hasOriginNode modelerNode ;
                        wflow:hasOriginArgument ac:o ;
                        wflow:hasVariable ModelDataVariable ;

ModelDataVariable
a    dcdm:LmtModel , wflow:DataVariable ;
dcdm:hasDomain dcdm:weather ;
dcdm:hasModelType DecisionTree.

classifierDataInputLink  a    wflow:InputLink ;
                        wflow:hasDestinationNode classifierNode ;
                        wflow:hasDestinationArgument ac:i ;
                        wflow:hasVariable TestDataVariable .
```

```

TestDataVariable    a    dcdm:Instance , wflow:DataVariable;
                    dcdm:notSameObject TrainingDataVariable;
                    dcdm:hasDomain dcdm:weather.

classifierOutputLink a    wflow:OutputLink ;
                    wflow:hasOriginNode classifierNode ;
                    wflow:hasOriginArgument ac:o ;
                    wflow:hasVariable ClassificationDataVariable .

ClassificationDataVariable a    wflow:DataVariable , dcdm:DtmClassification ;
                    dcdm:hasDomain dcdm:weather; // from the request

```

**Table 18: Relevant Excerpts of an Example Bound Specialized Workflow**

---

```

...
TrainingDataVariable
a    dcdm:Instance , wflow:DataVariable;
    dcdm:hasDomain dcdm:weather;
    wflow:hasDataBinding dcdm: weather-2007-31-101501.

TestDataVariable
a    dcdm:Instance , wflow:DataVariable;
    dcdm:hasDomain dcdm:weather;
    wflow:hasDataBinding dcdm:weather-2007-31-155754.

```

---

Other seeded specialized workflow candidates may generate different queries for finding data objects. For example, workflow candidates with BayesModeler or BayesClassifier will need DiscreteInstance as an input. A workflow with a NaiveBayesModeler and a J48Classifier will result in a query with the following data object descriptions:

```

TrainingDataVariable a    dcdm:DiscreteInstance , wflow:DataVariable;
                    dcdm:hasDomain dcdm:weather.

TestDataVariable    a    dcdm:Instance , wflow:DataVariable;
                    dcdm:notSameObject TrainingDataVariable;
                    dcdm:hasDomain dcdm:weather.

```

With a data catalog with four weather domain datasets (weather-2007-31-101501, weather-2007-31-101503, weather-2007-31-101656, and weather-2007-31-155754) that are all ContinuousInstances, the system will not find matching datasets for the workflows that need DiscreteInstances. In our running example, only 4 of the 18 candidate workflows with Lmt and J48 combinations (LmtModelerThenJ48Classifier, LmtModelerThenLmtClassifier, J48ModelerThen LmtClassifier, J48ModelerThen J48Classifier) will get results from the query to find matching data objects. For each candidate binding-ready workflow, the system produces twelve bindings since TrainingDataVariable and TestDataVariable should be bound to different weather datasets. That is a total of 48 candidate bound workflows generated in our running example. Table 18 shows an example of a bound workflow for LmtModelerThenJ48Classifier. For brevity, only the bindings and the DODs of the input data variables are shown.

The forward sweep sets all the parameter values of components and includes DODs for workflow data products. After that, the grounding step introduces data object identifiers for intermediate and final workflow data variables. Table 19 shows an example of a resulting workflow instance. The value of `maxJavaHeapSizeModelerParameterVar` is set in proportion to the size of the data sets that are bound to Training DataVariable (`dcdm:weather-2007-31-101501`). In particular, if the size of the data set is greater than 10,000 the parameter value is set to 1024M and if the size is less than 1,000 the value is set to 256M; otherwise it will be set to 512M.

**Table 19: A Workflow Instance After Workflow Instantiation**

---

```

LmtModelerThenJ48Classifier a wflow:WorkflowTemplate ;
  wflow:hasLink modelerTrainingDataInputLink , modelerJavaMaxHeapInputLink , classifierOutputLink ,
  modelerOutputClassifierInputInOutLink , modelerClassIndexInputLink , classifierDataInputLink ;
  wflow:hasNode classifierNode , modelerNode .

modelerNode a wflow:Node ; wflow:hasComponent ac:LmtModeler.
classifierNode a wflow:Node ; wflow:hasComponent ac:J48Classifier.

modelerTrainingDataInputLink a wflow:InputLink ;
  wflow:hasDestinationNode modelerNode ;
  wflow:hasDestinationArgument ac:d ;
  wflow:hasVariable TrainingDataVariable.

TrainingDataVariable a dcdm:Instance , wflow:DataVariable ;
  dcdm:hasDomain dcdm:weather ;
  wflow:hasDataBinding dcdm:weather-2007-31-101501.

maxJavaHeapSizeModelerParameterVar a wflow:ParameterVariable ;
  wflow:hasParameterValue "512M" ;

modelerJavaMaxHeapInputLink a wflow:InputLink ;
  wflow:hasDestinationNode modelerNode ;
  wflow:hasDestinationArgument ac:j ;
  wflow:hasVariable maxJavaHeapSizeModelerParameterVar .

modelerClassIndex a wflow:ParameterVariable .
  wflow:hasParameterValue 5 ; // from the request

modelerClassIndexInputLink a wflow:InputLink ;
  wflow:hasDestinationNode modelerNode ;
  wflow:hasDestinationArgument ac:i ;

modelerOutputClassifierInputInOutLink a wflow:InOutLink ;
  wflow:hasDestinationNode classifierNode ;
  wflow:hasDestinationArgument ac:m ;
  wflow:hasOriginNode modelerNode ;
  wflow:hasOriginArgument ac:o ;
  wflow:hasVariable ModelDataVariable .

ModelDataVariable a dcdm:BayesModel , wflow:DataVariable ;
  dcdm:hasModelType DecisionTree ;
  dcdm:hasDomain dcdm:weather ;
  wflow:hasDataBinding modelerOutputModelDataVariable_1191372118140.

classifierDataInputLink a wflow:InputLink ;
  wflow:hasDestinationNode classifierNode ;
  wflow:hasDestinationArgument ac:d ;
  wflow:hasVariable TestDataVariable .

TestDataVariable a dcdm:Instance , wflow:DataVariable ;
  dcdm:hasDomain dcdm:weather ;
  workflow:hasDataBinding dcdm:weather-2007-31-155754.

```

---

```

classifierOutputLink a wflow:OutputLink ;
wflow:hasOriginNode classifierNode ;
wflow:hasOriginArgument ac:o;
wflow:hasVariable ClassificationDataVariable .

ClassificationDataVariable a wflow:DataVariable , dcdm:DtmClassification ;
dcdm:hasDomain dcdm:weather; // from the request
wflow:hasDataBinding ClassificationDataVariable_1191372118140.

```

**Table 20: Example Ground Workflow Generated from a Workflow Instance**

---

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- generated: Tue Oct 02 17:42:29 PDT 2007 by Wings -->
<adag xsi:schemaLocation="http://www.griphyn.org/chimera/DAX http://www.griphyn.org/chimera/dax-1.10.xsd"
xmlns="http://www.griphyn.org/chimera/DAX"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.10" count="1" index="0" name="ModelerThenClassifier-dax198d8-b4d199239c5e7f99">

<!-- part 2: definition of all jobs -->
  <job id="Job1-04bd4f3cbfd2" namespace="http://www.isi.edu/ac/dm/library.owl" name="J48Classifier"
version="">
    <argument>-T <filename file="weather-2007-31-155754"/>
      -I <filename file="modelerOutputModelDataVariable_1191372118140"/>
      -O <filename file="ClassificationDataVariable_1191372118140"/> </argument>
    <uses file="modelerOutputModelDataVariable_1191372118140" link="input"/>
    <uses file="weather-2007-31-155754" link="input"/>
    <uses file="ClassificationDataVariable_1191372118140" link="output"/>
  </job>
  <job id="Job0-18917e3ec858" namespace="http://www.isi.edu/ac/dm/library.owl" name="LmtModeler"
version="">
    <argument>-Xmx 512M -t <filename file="weather-2007-31-101501"/>
      -d <filename file="modelerOutputModelDataVariable_1191372120250"/> -c 5 </argument>
    <uses file="weather-2007-07-31-101501" link="input"/>
    <uses file="modelerOutputModelDataVariable_1191372120250" link="output"/>
  </job>

<!-- part 3: list of control-flow dependencies (empty for single jobs) -->
  <child ref="Job1- 04bd4f3cbfd2">
    <parent ref="Job0-18917e3ec858"/>
  </child>
</adag>

```

---

The next step is ranking the 48 candidate workflows based on estimates of performance. For this, the predicted DODs for intermediate data sets are useful. For example, the size of intermediate data products is useful to obtain estimates on performance time for the different algorithms of the workflow components.

In the final step, each workflow instance is turned into a bound workflow that can be submitted to the workflow mapping and execution system. Table 20 shows the ground workflow extracted for the workflow instance in Table 18. It shows the format used by the Pegasus workflow mapping and execution engine [Deelman et al 05].

## 3.2 Workflow Ranking and Selection

For implementing the ranking module, System Research ported the HEFT [Topcuoglu 99] multiprocessor scheduling algorithm for scheduling on the Grid. The ranking for each workflow happens as follows:

- 1) For each job in the workflow, the weighted execution times are computed.
- 2) The downward ranks are computed then for each job. Downward rank of a job is defined as the longest distance from the root of the workflow to the job, excluding the computation of the job.
- 3) The jobs are then sorted in ascending order of their downward ranks.
- 4) Each of the above jobs in the sorted list, are then scheduled to a grid site. The grid site that is finally selected is the one that minimizes the finish time for the job.
- 5) The makespan of the workflow is determined as the maximum of the actual finish times of the leaf jobs of the scheduled workflow.
- 6) The higher the makespan of a workflow, the longer it will take to run. Thus, the workflow with the least makespan is the one with the highest rank, where higher the rank means better.

For the above algorithm, the SR relies on the Process Catalog for the following information:

- The sites where a particular job can be executed.
- The predicted performance of a code on a particular site, when run with the arguments specified in the DAX.

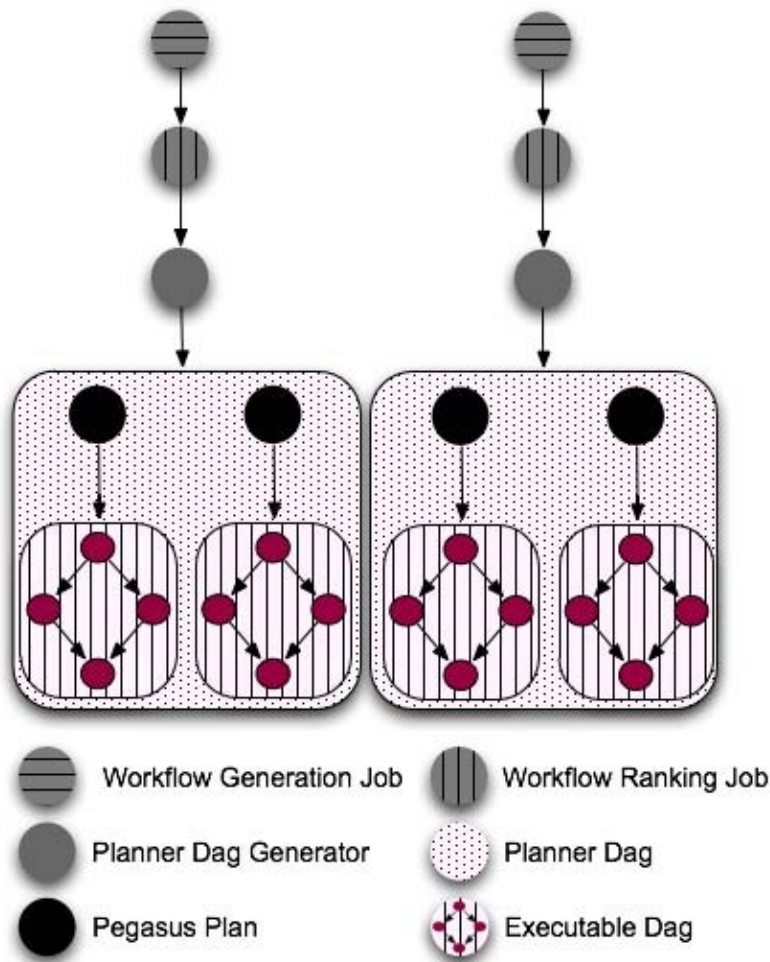
## 3.3 Managing Creation and Execution of Multiple Workflows

The Ensemble Manager (EM) component of the SR system is responsible for supporting the creation and the execution of multiple workflows at the same time.

Current workflow systems allow only sequential or uncoordinated creation and execution of a single workflow. The Ensemble Manager that we have developed coordinates and efficiently handles generation, ranking, planning and executing of 100's of workflows simultaneously on the TanGrid.

The EM takes a set of Tangram requests or Seeds. Each seed is specified with a configuration file that configures various aspects of how the request will be ranked, planned, executed etc.

The Ensemble Manager uses Condor and DagMan to manage the creation and execution of workflows. Instead of creating our own queuing mechanism, we built on the existing condor scheduler. By generating appropriate condor jobs to do the creation and execution steps and adding the dependencies in a dag, the Ensemble Manager can efficiently execute 100's of workflows simultaneously in an efficient manner. EM extends and uses several Condor features like job priority, job start time, etc. to provide various features like start time for each workflow in an Ensemble, different priorities across each workflow, wall time by which the workflow needs to be finished etc. EM uses the postscript feature of Condor that allows it to run a script when a job finishes, no matter if the parent job failed or succeeded, to monitor the successful completion of different creation, planning and execution jobs.



**Figure 6: Structure of the DAG constructed by the Ensemble manager**

#### Ensemble Manager Features:

- Allows submission of a set of seeds (a portfolio)
- Each seed is specified with its own seed configuration file
- Each seed can have a different priority
  - The priority is applied to all stages of the workflow generation and execution
- Each seed can have a different start time
- Each seed can have a different wall-time or end time. If the workflow is still executing when the wall-time finishes the workflow is killed
- Continuously monitors the progress of the portfolio and seeds and updates the status of the seeds in a database.

The Ensemble Manager takes in an input file called portfolio containing Seeds and seed configurations. From these seed configurations the ensemble manager generates several configuration files for workflow generation, ranking and planning. After generating these configuration files, the EM creates the main ensemble dag and submits files required to generate, rank, plan and execute the given seeds in the portfolio.

Figure 6 shows the structure of the DAG constructed by the Ensemble manager for a single portfolio (EM Request) with 2 Requests or Seeds (Workflows).

For each seed the EM creates the following jobs:

1. Workflow Generation Job: This job takes in the Seed identifier and generates the appropriate Abstract Workflows (DAX)
2. Workflow Ranking Job: This job takes in the DAX generated by the earlier Ranking Job and ranks the generated DAX's and produces a ranked DAX file.
3. Planner DAG Generator Job: The third job added parses the generated ranked file and generates a planner DAG to plan the ranked workflows and execute them
4. Planner DAG: The planner dag consists of
  - a. Pegasus Plan Job: This job ranks one of the ranked DAX's in the workflow and produces an executable workflow to execute on the TanGrid
  - b. The Executable Workflow DAG and submit files: This Workflow has the jobs which create the Knowledge Base in the Allegro Graph, the transfer job that copies the data from the Evidence Data Base to the Knowledge Base, jobs that run the wrapped executables etc.

## 4. SUPPORT FOR PROGRAM INTEGRATION

This section describes SR's effort on developing workflow generation API and supporting program integration. The API describes all the queries to be issued by SR's automated workflow generation and execution capability to other architectural components, notably AC and DC.

## 4.1. SR-12 Workflow Generation API

The workflow generation API defines the invocation of DC and PC functions during the major steps in workflow generation and execution.

### Step 0: Initiating the Workflow Generation Process

The workflow request may just be to run a workflow template for some purpose or run a variant of an existing template.

```
SEA->SR: solveWorkflowRequest( $WT_i$ ,  $DOD_r$ )
SR->SEA: { {dataSetIDi} }
```

$WT_i$  ==> workflow template id

$DOD_r$  ==> data object descriptions from the request given in the teo/data/metrics/dataAccess namespaces on some data variables

### Step 1: Find Candidate Analyses

This step was not needed for SR-12, since the initial request will specify a template and associated constraints.

### Step 2: Find Data Object Descriptions and Argument Mappings

#### [Q2.1]

```
SR->AC: findInputDataRequirements( $C_i$ , { $DOD_i$   $AM_i$ })
AC->SR: {  $C_j$  { $DOD_k$   $AM_k$ } }
```

$C_i$  ==> an abstract or concrete component description from the adl namespace from a workflow template

{ $DOD_i$ ,  $AM_i$ } ==> data object descriptions in the teo, data, and metrics namespace for the output of  $C_i$  and component-argument-to-template-dataVariable mappings for  $C_i$  - dataVariables will be in the *sr* namespace and the component arguments will be in the *adl* namespace. Arguments refer to the data object inputs and outputs of the component. Note that the DOD's contain information about patterns and parameters.

$C_j$  ==> a component description from the adl namespace for concrete component

{ $DOD_k$ ,  $AM_k$ } ==> data object descriptions in the teo, data, and metrics namespace for the inputs & outputs and component-argument-to-template-dataVariable mappings for  $C_j$  - dataVariables will be in the *sr* namespace and the component argument will be in the *adl* namespace. Note that the DOD's contain information about patterns and parameters.

### Step 3: Data Source Selection

#### [Q3.1]

```
SR->DC: FindDataSources({ $DODi$ })
DC->SR: { $DVj$   $DSj$ }
```

$\{DOD_i\} \implies$  data object descriptions from the `teo/data/metrics/dataAccess` namespace mapped to data variables (in the `sr` namespace) from a workflow template. These descriptions explain constraints on and between the data variables in a particular workflow template

$\{DV_j DS_j\} \implies$  data variables (in the `sr` namespace) mapped to data source ids (in the `dc` namespace).

#### Step 4: Workflow Instance Generation

##### [Q4.1]

SR->DC: FindDataMetricsForDatasource( $DS_i$  &opt {*dc metric or characteristic*})

DC->SR:  $DOD_i$

$DS_i \implies$  A data source id

{*dc metric or characteristic*}  $\implies$  specific data metric or data characteristic - if not provided the full set of metrics and characteristics are returned.

$DOD_i \implies$  Data object descriptions in the `teo`, `data`, and `metrics` namespace for the  $DS_i$  data source

##### [Q4.2]

SR->AC: FindOutputDataPredictedDescriptions( $C_i$ , { $DOD_i$   $AM_i$ })

AC->SR: { $DOD_k$   $AM_k$ }

$C_i \implies$  a component description in the `adl` namespace for a concrete component

{ $DOD_i$ ,  $AM_i$ }  $\implies$  data object descriptions in the `teo`, `data`, and `metrics` namespace for the input and output of  $C_i$  and component-argument-to-template-dataVariable mappings for  $C_i$  - dataVariables will be in the `sr` namespace and the component arguments will be in the `adl` namespace. Arguments refer to the data object inputs and outputs of the component. Note that the `DOD`'s may contain information about patterns and parameters.

{ $DOD_k$   $AM_k$ }  $\implies$  data object descriptions in the `teo`, `data`, and `metrics` namespace for the input and output of  $C_i$  given { $DOD_i$   $AM_i$ }

##### [Q4.5]

SR->AC: GetInvocationCommand( $C_i$ , { $DOD_i$   $AM_i$ })

AC->SR: { $A_j$ }

$C_i \implies$  a component description in the `adl` namespace for a concrete component

{ $DOD_i$   $AM_i$ }  $\implies$  data object descriptions in the `teo`, `data`, and `metrics` namespace for the input and output of  $C_i$  and component-argument-to-template-dataVariable mappings for  $C_i$  - dataVariables will be in the `sr` namespace and the component arguments will be in the `adl` namespace for input and output data objects. Arguments refer to the data object inputs and outputs of the component.

$A_j \implies$  an argument list as specified in the argument section of the [Basic Component Encapsulation](#) page. All parameter arguments must be returned with actual values.

## Step 5: Data Query and Workflow Reduction

Query 5.1a is made to the DC to get the locations of datasets and any attributes associated with it.

Query 5.2b is made to the Pattern Catalog to get the locations of patterns and attributes. The query and results look similar for both queries.

### [Q5.1a]

```
SR->DC: FindDataSetLocationAndAttribs( $DS_i$ )
DC->SR: { $DSL_A$  | NULL}
```

$DS_i \implies$  a data source id

{ $DSL_A$ }  $\implies$  data source locations and access protocols for  $DS_i$

{NULL}  $\implies$  If NULL then the datasetid does not exist

### [Q5.1b]

```
SR->AC: FindPatternLocationsAndAttribs( $PS_i$ )
AC->SR: { $PSLA$  | NULL}
```

$PS_i \implies$  a pattern source id

{ $PSLA$ }  $\implies$  pattern source locations and access protocols for  $PS_i$  {NULL}  $\implies$  pattern source locations for pattern id does not exist

### [Q5.2a]

```
SR->AC: GetPredictedPerformance( $C_i$ , { $DOD_i$   $AM_i$ }, &opt architecture)
AC->SR: {architecture PP }
```

$C_i \implies$  a component description in the adl namespace for a concrete component

{ $DOD_i$   $AM_i$ }  $\implies$  data object descriptions in the teo, data, and metrics namespace for the input of  $C_i$  and component-argument-to-template-dataVariable mappings for  $C_i$  - dataVariables will be in the sr namespace and the component arguments will be in the adl namespace. Arguments refer to the data object inputs and outputs of the component.

*architecture*  $\implies$  a specific hardware architecture

{*architecture PP* }  $\implies$  architecture and predicted performance for  $C_i$  given { $DOD_i$   $AM_i$ }

### [Q5.2b]

```
SR->AC: GetPredictedPerformance( $C_i$ , { $DOD_i$   $AM_i$ }, &opt site)
AC->SR: {site PP}
```

$C_i \implies$  a component description in the adl namespace for a concrete component

$\{DOD_i AM_i\} \implies$  data object descriptions in the teo, data, and metrics namespace for the input of  $C_i$  and component-argument-to-template-dataVariable mappings for  $C_i$  - dataVariables will be in the sr namespace and the component arguments will be in the adl namespace. Arguments refer to the data object inputs and outputs of the component.

$site \implies$  a cluster or part of a cluster with homogenous arch/os/glibc (speed and memory may/can be different)

$\{site PP\} \implies$  site and predicted performance for  $C_i$  given  $\{DOD_i AM_i\}$

### Step 6: Workflow Ranking

Not used in SR-12.

### Step 7: Workflow Mapping

#### [Q7.1]

```
SR->AC: FindCodeLocations ( $C_i$ )
AC->SR:   $\{L_i\}$ 
```

$C_i \implies$  a component description using the adl namespace

$L_i \implies$  sites where  $C_i$  is located , type of component, and system information (arch,os,os version, glibc)

#### [Q7.2]

```
SR->AC: GetDeploymentRequirements( $C_i$ ,  $site$ )
AC->SR:   $\{R\}$ 
```

$C_i \implies$  a component description in the adl namespace for a concrete component

$site \implies$  a cluster or part of a cluster with homogenous arch/os/glibc (speed and memory may/can be different)

$\{R\} \implies$  requirements for running  $C_i$  on  $site$

### Step 8: Workflow Execution

#### [Q8.1]

```
SR->DC: AddDataSetIDs( $DS_i$ ,  $DSL A_i$ )
DC->SR:   $\{t \mid nil\}$ 
```

$DS_i \implies$  a data source id

$DSL A_i \implies$  data source locations,access protocols and attribs for  $DS_i$

This is related to the 5.1 query above. The information registered in 8.1 is available for queries during the generation of other workflows during step 5 using 5.1

## [Q8.2]

SR->DC: FindActualDataSetCharacteristics( $DS_i$  &opt {*dc metric or characteristic*})

DC->SR: *DMC*

$DS_i$  ==> A data source id

{*dc metric or characteristic*} ==> specific data metric or data characteristic - if not provided the full set of metrics and characteristics are returned.

*DMC* ==> data metrics and/or characteristics for the data source indicated by  $DS_i$

## 5. MAJOR MILESTONES: DEMONSTRATIONS AND EVALUATIONS

This section describes SR's major milestones throughout the program.

### 5.1 SR-6 Demonstration

The objectives of SR-6 demonstration were:

- 1) Demonstrate that SR can create and execute program-relevant workflows drawing from pre-existing SR technologies in a few months time. This serves as a proof of concept that:
  - A Tangram system will result in significant time savings over manually constructed workflows
  - A Tangram system will enable workflows that are much more complex than the at-most 3 steps/algorithms workflows that analysts develop today.
- 2) Demonstrate that an efficient inter-algorithm data exchange mechanism can be implemented and integrated with the SR workflow system. The algorithm interface will be implemented by GU through the first instantiation of GU's Gather subsystem.

The first objective was accomplished by the SR team on December 14, 2006. The workflows for this demonstration were constructed by hand from five algorithms and two synthetic data sources. The second objective was accomplished on February 26, 2007. GU provided workflow components for translating data in and out of algorithms, the data and algorithms were provided by SR from the December 2006 workflows.

Figure 7 shows the workflows used in December 2006. The 2007 workflows are isomorphic to Workflow 2 (W2) but they used the Gather translators.

The workflows designed for SR-6 were designed to be appropriate for a data-to-warning system, as envisioned by Tangram. The SR-6 workflows were designed with no guarantees as to the quality or accuracy of the results.

The SR-6 demonstration used the Hats simulator to produce synthetic data sources to demonstrate a data to hypothesis to warnings system. The Hats simulator produces a variety of data grouped into 22 files (See “Hats Simulator Batch Data Sets”, J. Moody, Nov 2, 2006). The workflow uses only five of these possible data sources:

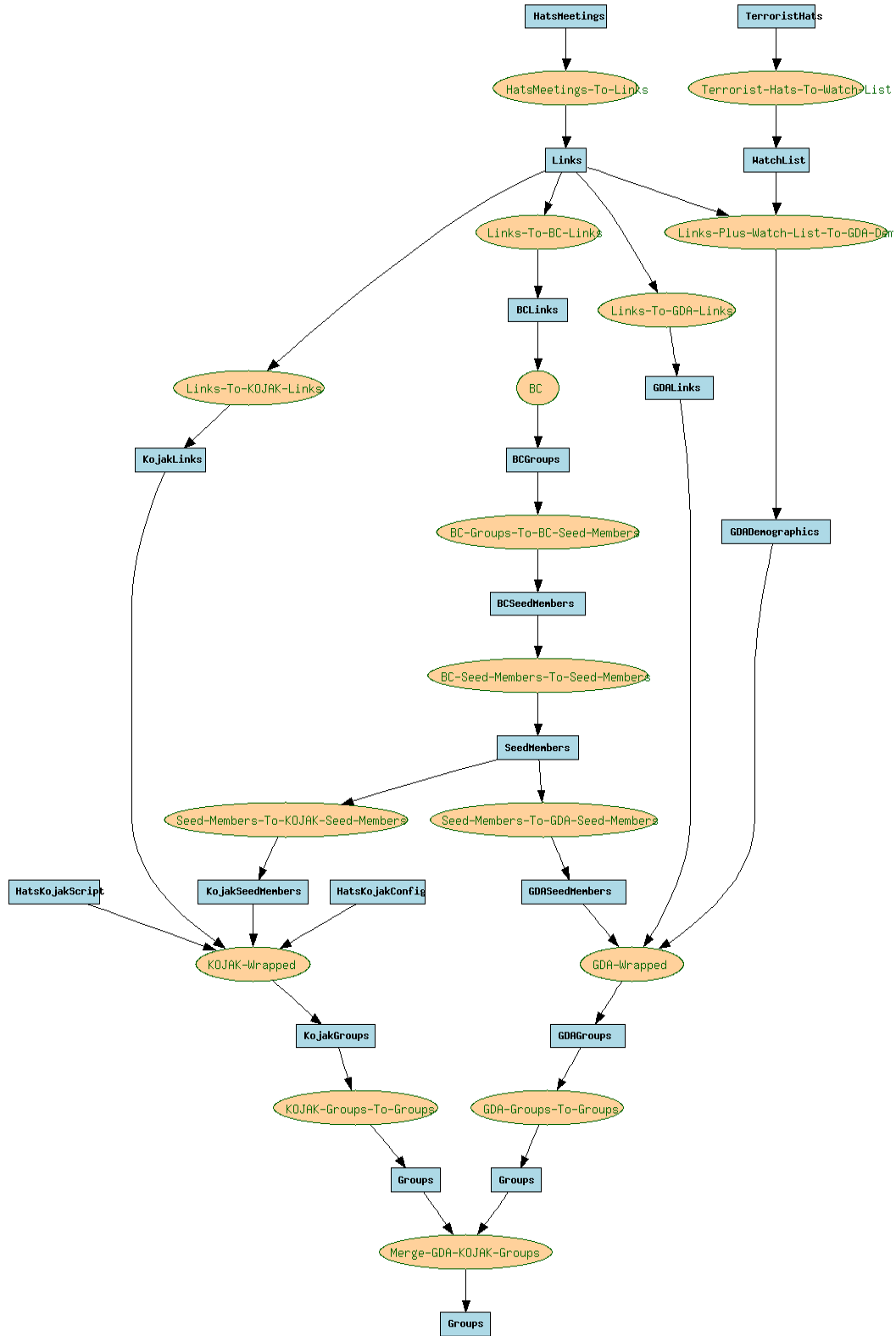
- meetings: meetings between individual hats that occurred during the simulation.
- watch list: a list of hats that are known to be malicious.
- capabilities: a set of capabilities that exist in the simulation, either because they are required to attack some beacon or because some hat has them.
- trades: exchanges of capabilities between two hats that occurred in the simulation.
- beacon vulnerabilities: for each beacon it specifies the capabilities required to attack it (reflecting the vulnerabilities of that beacon).

The workflows include Eagle algorithms, and to use them we had to turn legacy code into workflow components. The components used include

- GDA: group finder (CMU, Kubica)
- BC: group finder (Newman, ISI implementation: Moody)
- KOJAK: group finder (ISI, Adibi, Chalupsky)
- RLP: suspicion scorer (ISI, Galstyan)
- NetKit: suspicion scorer (NYU, Macskassy)
- CapTracker: non-linear tracking (ISI, Mitra, Galstyan)

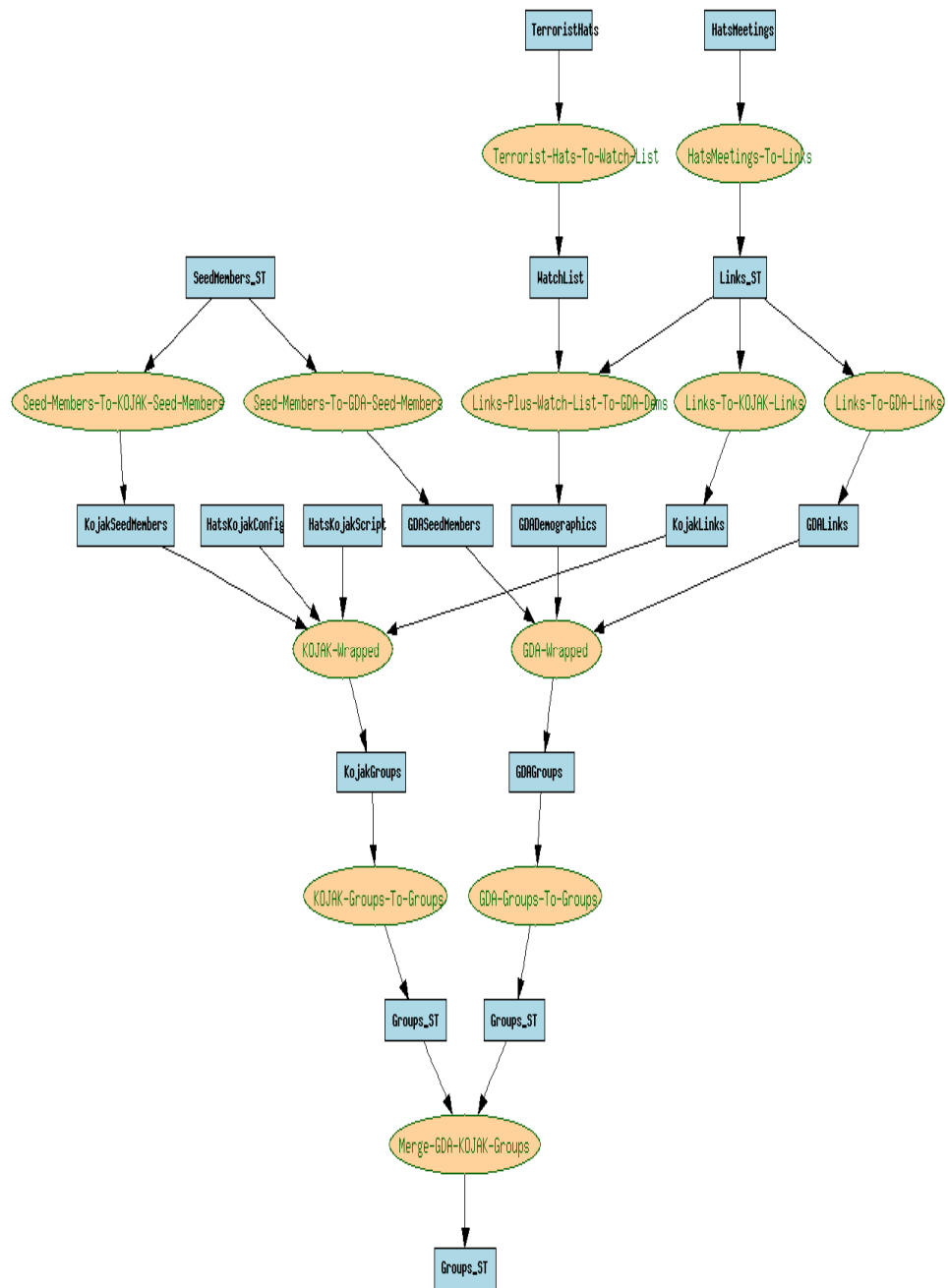
Several additional components were developed by SR.

- merge: merging the results of  $n$  components
- filter: selecting data views, heuristic selection of content
- translate: selecting data views, format conversion



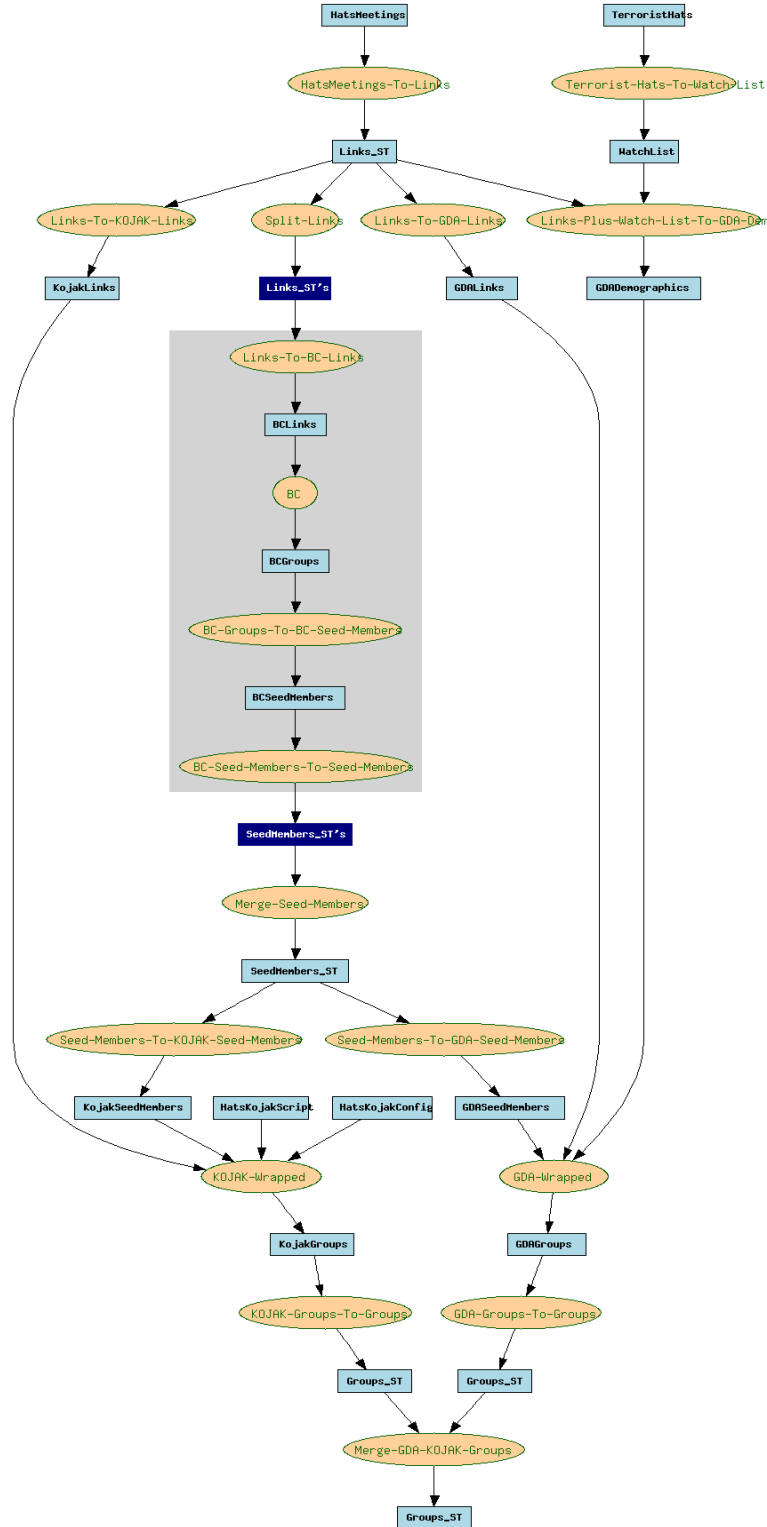
W1: Search for group seed members then run two group finding algorithms then merge results

**(a) Original workflow**



W2: W1 with group seed members coming from primary sources

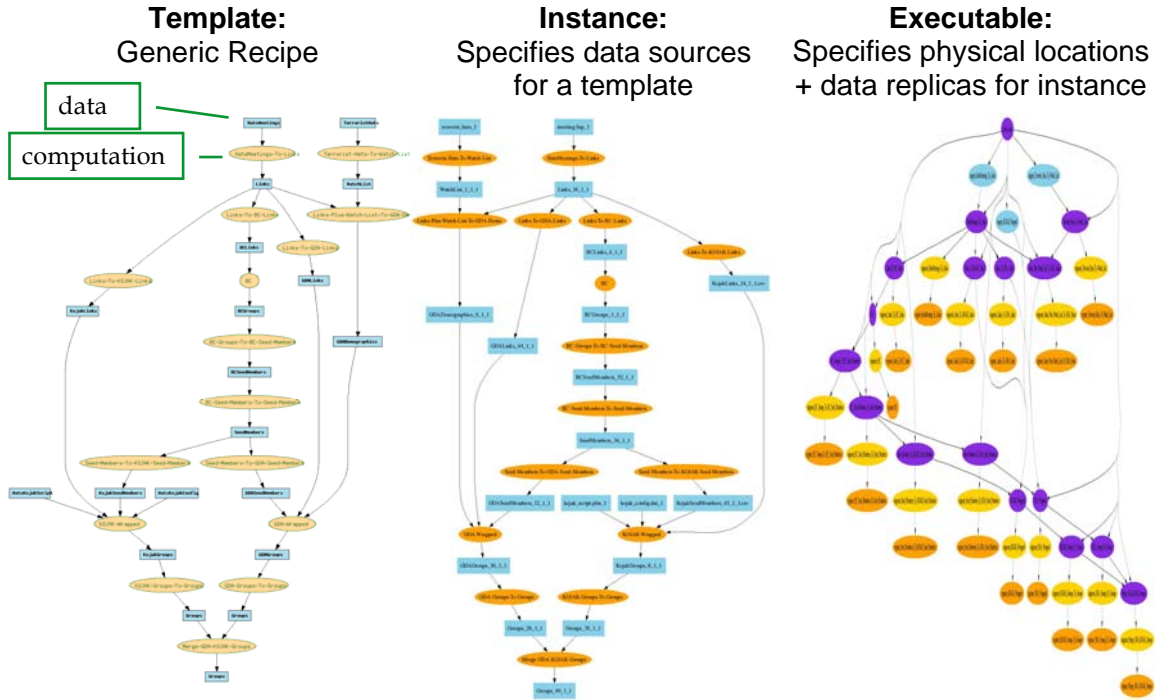
**(b) ManualSeed:** Seed groups are provided



W3: W1 with efficient execution by parallel search for group seed members

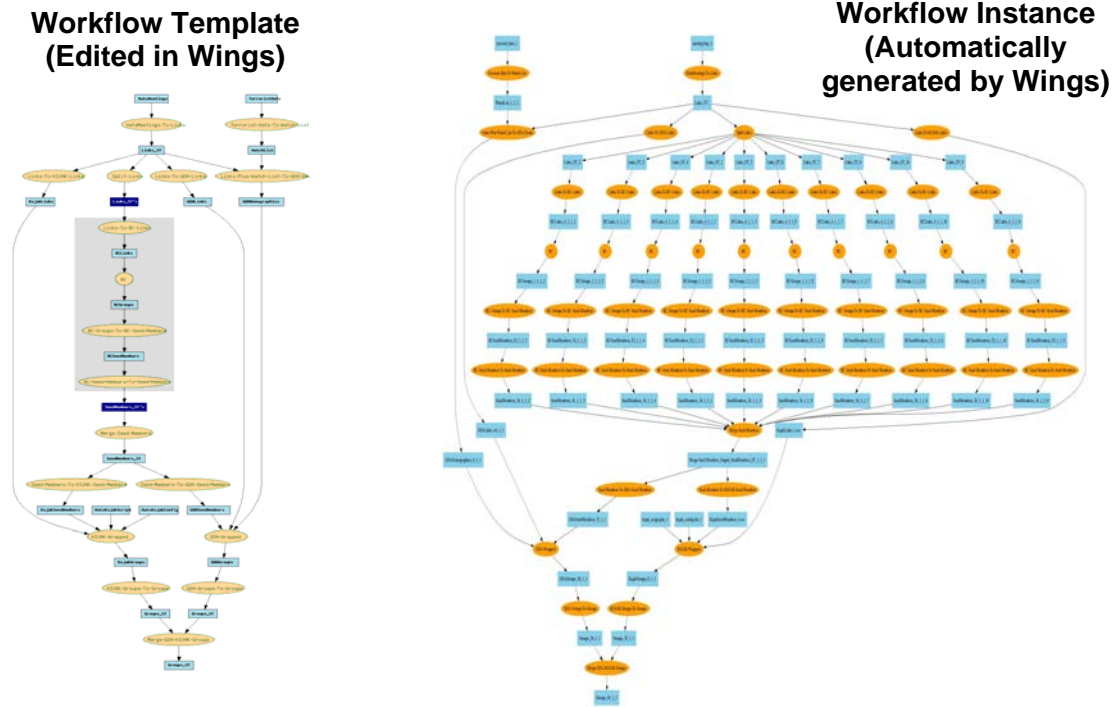
(c) **ParallelSeed**: Seed groups are found through parallel computation

**Figure 7: Workflows Demonstrated in December 2006**



**Figure 8: Workflow Creation in SR-6 from Template to Instance to Execution**

Figure 8 shows an example template, workflow instance and executable workflow from SR-6. We executed the first Tangram-relevant workflow using the workflow system and the program testbed. It is a portion of the planned SR-6 workflow focused on group detection. It used a small data set from Hats (about 500 hats), and was run on the ISI Skynet cluster that was part of the program testbed at that time. It contains 16 executable components. After a few weeks of design, code development, and code characterization, it took a few hours to create the workflow template with the Wings editor, and a few hours to generate the executable workflow. Note that the structure of the above workflow is very simple (by design), so the nodes and files in the template and the instance mirror one another. Figure 9 shows a workflow with parallelized seed member formation where this is not the case.



(a) Parallel Seed: Workflow Instance Creation

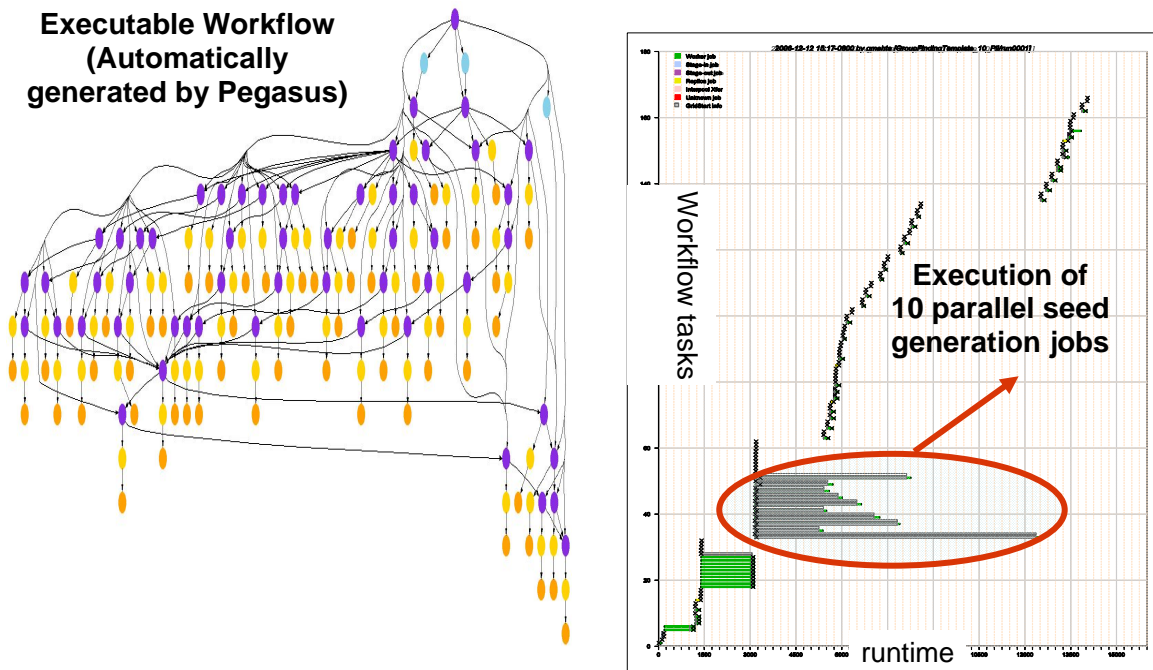
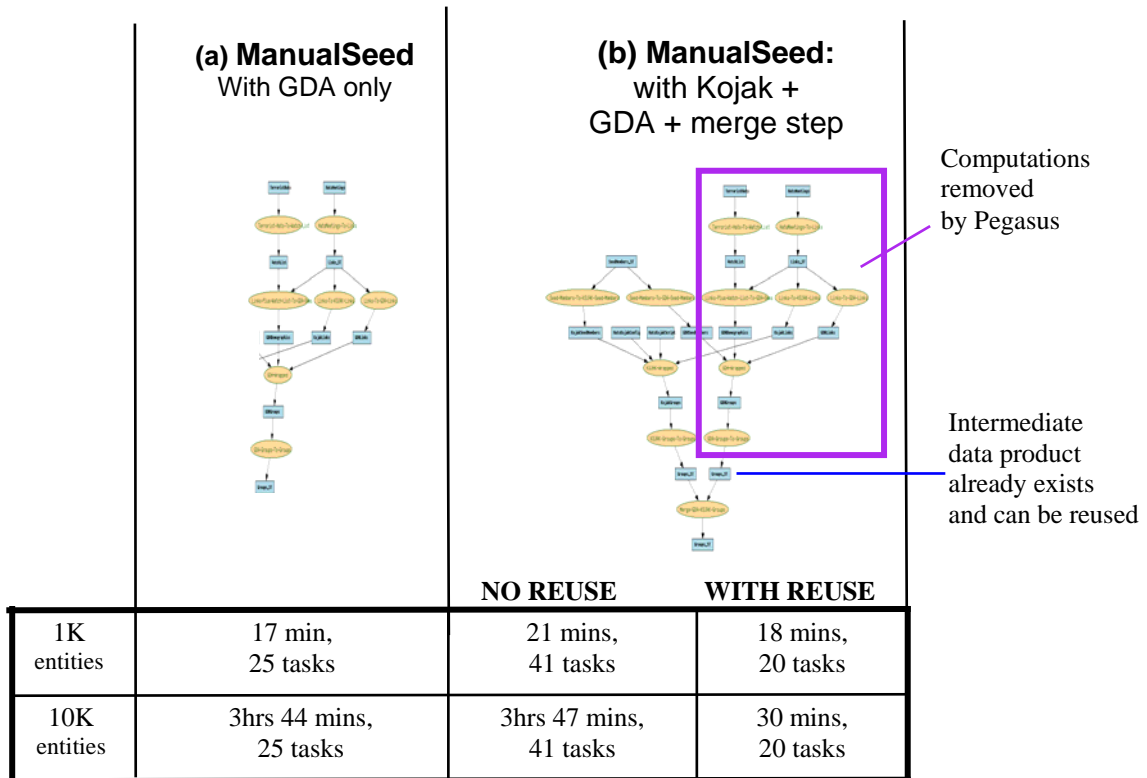


Figure 9: Example Workflow with Parallel Seed Creation



**Figure 10 : Supporting Data Reuse Across Workflows**

The Pegasus Workflow Planner reuses existing data products on the grid to refine and reduce the workflow so that only those computations required are run. The existing data products are transferred from the remote sites to the execution locations. Figure 10 shows how data reuse makes computation more efficient. The workflow on the right (b) has some of the same computation as the workflow on the left (a). When workflow (a) is run first, with reuse, the system can save the computations needed for the same portion.

In summary, the SR-6 demonstration provided a proof of concept of the Tangram system and a set of workflows to guide SR-12 and SR-18 integration discussions.

## 5.2 SR-12 Demonstration

The SR-12 demonstration was designed by the SEA evaluation team.

The SR team contributed:

- A workflow reasoning system that can query the Data Characterization and Process Characterization services to properly select the data sources and workflow components for a given workflow request to generate workflow instances. To this end, the SR team:
  1. designed a workflow generation algorithm to automatically create executable workflows from high-level workflow requests
  2. defined APIs for DC and PC services to be invoked during that process
  3. implemented a workflow system that had these capabilities
- A workflow execution system that can execute the workflow instances that enable workflow components to interact with source evidence and produce hypotheses that will be used by downstream workflow components using a unified representation provided by Graph Unification. To this end, the SR team:
  - designed a specification to convert executable codes into proper workflow components using a Basic Component Encapsulation schema and methodology

SR-12 demonstrated automated workflow generation using externally provided data and component catalogs on Tangram-relevant workflows. The SR team developed a **Workflow Generation API** that specifies how the workflow generation system interacts with the systems being developed by other program participants.

SR's workflow reasoning system queries the Data Characterization and Process Characterization services to properly select the data sources and workflow components for a given workflow request to generate workflow instances. SR's workflow execution system executed the workflow instances that enable workflow components to interact with source evidence and produce hypotheses, which were used by downstream workflow components using a unified representation provided by Graph Unification.

The following subsections describe SR-12 functional and technical requirements defined by the SEA evaluation team that are relevant to SR.

### 5.2.1 SR-12 Functional Requirement Relevant to SR

This section summarizes SR-12 functional requirements (FR) defined by the SEA evaluation team that are relevant to SR.

#### Functional Requirement 2 (FR-2)

FR-2: The SR-12 System shall demonstrate the use of more than 2 workflow components in the workflow (Evaluate the GU inter-component data exchange process)

SEA/SR Claim: Workflow templates contain 3 classes of Workflow Component: Group Detection Process, Pattern Matching Process, Data Union Process

The workflow template used in the SR-12 workflow requests uses 3 types of workflow components: GroupDetection, DataUnion, and PatternMatching.

### **FR-3**

FR-3: The SR-12 System shall demonstrate the ability to orchestrate workflow components of more than one process type (Evaluate GU ability to exchange data)

SEA/SR Claim: Workflow templates contain 3 classes of Workflow Component: Group Detection Process, Pattern Matching Process, Data Union Process

The workflow template used in the SR-12 workflow requests uses 3 types of workflow components: GroupDetection, DataUnion, and PatternMatching.

### **FR-4**

FR-4: The SR-12 System shall demonstrate the ability to instantiate a given workflow template in multiple ways, depending on the characteristics of the available dataset(s), the availability of suitable patterns, and/or other conditions (Evaluate the work flow composition process based on variances in data, search patterns or other work flow pre-conditions)

SR Claim: The workflow system invokes PC and DC services with constraints imposed by the workflow template and the workflow request

SR invoked PC and DC through the following API queries:

- Q2.1 was invoked to specialize components in the workflow template. For the workflow request shown above, this query was invoked 3 times and 5 components were returned, resulting in the generation of 4 workflow candidates from the original workflow.
- Q3.1 was invoked to find valid data sources for the request. For the workflow request shown above, this query was invoked 4 times, each call returning 3 bindings for the workflow, resulting in the generation of 12 workflow candidates.

### **FR-5**

FR-5: The SR-12 System shall demonstrate handling of multiple workflow requests concurrently

(Evaluate the workflow manager's ability to accept multiple concurrent work flow requests from TEE)

SR Claim: The workflow system can manage the workflow generation process for several requests submitted.

Several alternative workflow requests were created and submitted concurrently to the workflow system.

## **FR-6**

FR-6: The SR-12 System shall demonstrate the ability to track lineage, pedigree, and provenance of assertions and hypotheses during workflow processing (Evaluate the Tangram system's ability to acquire and store information about each workflow instance)

SR Claim: The workflow system records extensive traces of the workflow generation process (including queries issued and responses obtained, workflow candidates generated or eliminated) and of the workflow execution process. Log4J is used for the workflow generation phases.

A detailed trace of the generation of executable workflows for one of the requests is uploaded in the code repository.

### **5.2.2 SR-12 Technical Requirements Relevant to SR**

This section summarizes SR-12 technical requirements (TR) defined by the SEA evaluation team that are relevant to SR.

#### **Technical Requirement-2 (TR-2)**

TR-2: The SR-12 System shall demonstrate the ability to execute multiple workflow instances concurrently (Evaluate the ability to plan and deploy multiple concurrent workflow instances)

SR Claim: SR system (via Ensemble Manager and Pegasus?) will be capable of scheduling and initiating concurrent executions of workflow instances on TanGrid.

SR Claim: SR system (via Ensemble Manager and Pegasus?) will ensure no node is used for more than one Workflow Component execution at a time (due to GATHER implementation limitation)

#### **TR-3**

TR-3: The SR-12 System shall demonstrate the ability to tailor/scope workflow instances on the basis of characteristics of available resources (Evaluate SR (Pegasus) mapping of workflow processes to hardware resources based on PC process installation information)

SR Claim: Pegasus will select available nodes on the basis of hardware requirements provided by PCat for each Workflow Component

#### **TR-4**

TR-4: The SR-12 System shall encode and interpret workflow templates referencing process types available via the Process Catalog ontologies (e.g. Capability Layer of ADL) (Evaluate SR's ability to construct workflows consistent with the PC definition of process types)

SR Claim: Workflow templates created by hand referencing terms available in PC process ontology

The workflow used in the request above used the following classes from the process ontology namespace: GroupDetectionProcess, DataUnionProcess, and PatternMatchingProcess.

## **TR-5**

TR-5: The SR-12 System shall use the Process Catalog in the course of instantiating and executing the workflow

SR Claim: Workflow queries PCat to generate workflows via Q2.1, Q4.2, Q4.5, Q5.2, Q7.1, Q7.2

Q2.1 to PC was invoked 5 times, Q4.2 to PC was invoked 36 times, and Q4.5 to PC was invoked 36 times.

SR Claim: Workflow system able to generate candidates and create executable DAXes on the basis of PCat responses for Q2.1, Q4.2, Q4.5, Q5.2, Q7.1, Q7.2

Q2.1 to PC was used to create 4 workflow candidates, Q4.2 and Q4.5 to PC were used generate executable DAXes for 12 workflow candidates.

## **TR-6**

TR-6: The SR-12 System shall use the Data Characterization service to determine which workflow template to execute

SR Claim: Workflow queries DMS to generate workflow candidates via Q3.1, Q4.1, Q5.1a

Q3.1 to DC was invoked 4 times, Q4.1 to DC was invoked 4 times.

SR Claim: SR able to create executable DAX's on the basis of PCat responses for Q3.1, Q4.1, Q5.1a

Q3.1 to DC was used to create 12 workflow candidates, and Q4.1 was used in the generation of DAXes.

## **TR-11**

TR-11: The SR-12 System shall demonstrate the ability to exchange data between workflow components using the GU graph specification

SR Claim: SR components (via Pegasus / GridFTP) can physically move files produced on one processing node to another

## 5.3 SE-18 Evaluation

SR's contributions to the SE-18 Evaluations include:

- Data Reuse: SR invokes DC services for registering workflow data products including intermediate products. The extended workflow generation API (Q4.3a in particular) allows reuse of existing data to save computation
- A workflow ranking algorithm that takes into account ingest time for input data sources. The ranking algorithm is described in Section 3.2.
- The Ensemble Manager (EM) system was developed to support multiple concurrent workflow runs. More information about EM is available below in Section 3.3.
- SR provided detailed Workflow System Logging records.
- The Tangram Grid and software deployment are supported by SR. SR provided documentation to implement an automated site catalog as well as instructions on how to install the Tangrid Components and SR Functions.
- An extended Workflow Generation API: Workflow Generation API Version 2.1 was developed and released for the SE-18 evaluation.

Figure 1 highlighted the new SR capabilities developed for SE-18. The following summarizes the SE-18 metrics and requirements relevant to SR.

### 5.3.1 SE-18 Metrics and Requirement Relevant to SR

SR implemented capabilities including automatic workflow generation, workflow ranking, and ensemble manager support the following SE-18 metrics and requirements.

- FM-WF-A-1: Manual construction of a template by a person intending to translate an analytic line of inquiry into an abstract workflow template. Per SEA's request, SR provided a brief document on how to write requests
- FM-WF-A-2: Time to validate / generate candidate workflow instances Not to Exceed (NTE) 2 minutes. Per PMO's request, SR has created a discussion page on the workflow generation time metric (FM-WF-A-2). The newly developed Ensemble Manager manages all workflow generation steps (steps 1-8).
- FM-WF-A-3: Time to set up workflow ranking experiments NTE 1 hour. For this SR have developed extensions to workflow ranking (step 6 of workflow generation algorithm)

- FM-WF-A-4: Time to collect experiment metrics NTE 120 hours. SR's design of logging ontology and format supports this.
- FM-HW-A-1: Deploy Tangram Function on new hardware to include it in the Tangrid NTE 2 days. SR's Tangram Grid deployment support includes instructions to deploy Tangram Grid Functions on a new hardware
- FM-HW-M-1: Any changes to hardware configurations for a node in TanGrid are reported / captured by any resource requiring an understanding of hardware configuration / capabilities. SR's Tangram Grid deployment support includes documentation to implement probes to update an automated Site Catalog.
- FM-SW-M-1: Changed Tangram Component: NTE 30 minutes to upgrade / reinstall, configure, and make operational any releasable updates to a Tangram System Component. SR's Tangram deployment support instructions on how to build/install/configure Tangrams SR Components .
- SE18-FR1: The System's execution shall be fully automated. This is supported by the existing automated workflow generation algorithm.
- SE18-FR2: The System shall operate continuously. SR's Ensemble Manager can be configured to support continuous operation.
- SE18-FR8: For SE-18, at least 50 distinct lines of inquiry can be running concurrently (for SE-18, a line of inquiry will be equivalent to a workflow request). The Ensemble Manager handles multiple requests.
- SE18-FR9: The System shall log all data accesses, to include at least time of access, LOI, workflow instance in order to provide a full audit log of data-related activities conducted by the System. The new Workflow System Logging supports necessary audit log.
- SE18-FR10: The System shall enable the automated introduction and characterization of new workflow components to a Tangram System environment without interrupting surveillance and warning functions. This is supported by the automated workflow generation algorithm.
- SE18-FR11: The System shall enable the automated characterization and ingestion of newly identified data sources without interrupting surveillance and warning functions. This is supported by the automated workflow generation algorithm.
- SE18-FR12: The System shall enable the automated inclusion and instantiation of new workflow requests without interrupting surveillance and warning functions. The Ensemble Manager handles multiple concurrent requests.

- SE18-FR13: The System shall enable the inclusion of new hardware into its operating environment without interrupting surveillance and warning functions. SR's Tangram Grid deployment support includes probes to populate Site Catalog.
- SE18-FR14: The System shall enable the automated introduction and characterization of modified workflow components to a Tangram System environment without interrupting surveillance and warning functions. This is supported by the automated workflow generation algorithm.
- SE18-FR15: The System shall enable the automated characterization and ingestion of modified data sources without interrupting surveillance and warning functions. This is supported by the automated workflow generation algorithm.
- SE18-FR16: The System shall enable the automated inclusion and instantiation of modified workflow requests without interrupting surveillance and warning functions. The Ensemble Manager handles multiple concurrent requests.
- SE18-FR17: The System shall enable the removal of existing execution hardware from its operating environment without interrupting surveillance and warning functions. This is supported by the existing workflow mapping algorithm.
- SE18-TR1: The SE-18 System shall encode and interpret workflow templates referencing process types available via the Process Catalog ontologies (e.g. Capability Layer of PDL). This is supported by the automated workflow generation algorithm.
- SE18-TR2: The SE-18 System shall demonstrate the ability to accept multiple, distinct workflow request concurrently. The Ensemble Manager handles multiple concurrent requests.
- SE18-TR3: The SE-18 System shall demonstrate the ability to generate and rank workflow instances from multiple workflow requests. SR extended the workflow ranking (step 6 of workflow generation algorithm) to rank workflow instances from the same workflow request.
- SE18-TR4: The SE-18 System shall demonstrate the ability to tailor/scope workflow instances on the basis of characteristics of available hardware resources. This is supported by the existing workflow mapping algorithm.
- SE18-TR5: The SE-18 System shall use the Process Catalog in the course of instantiating the workflow. This is supported by the automated workflow generation algorithm.
- SE18-TR14: The SE-18 System shall demonstrate the ability to track pedigree (source identification) of assertions and hypotheses during workflow processing. The Workflow System Logging will support necessary audit logging.

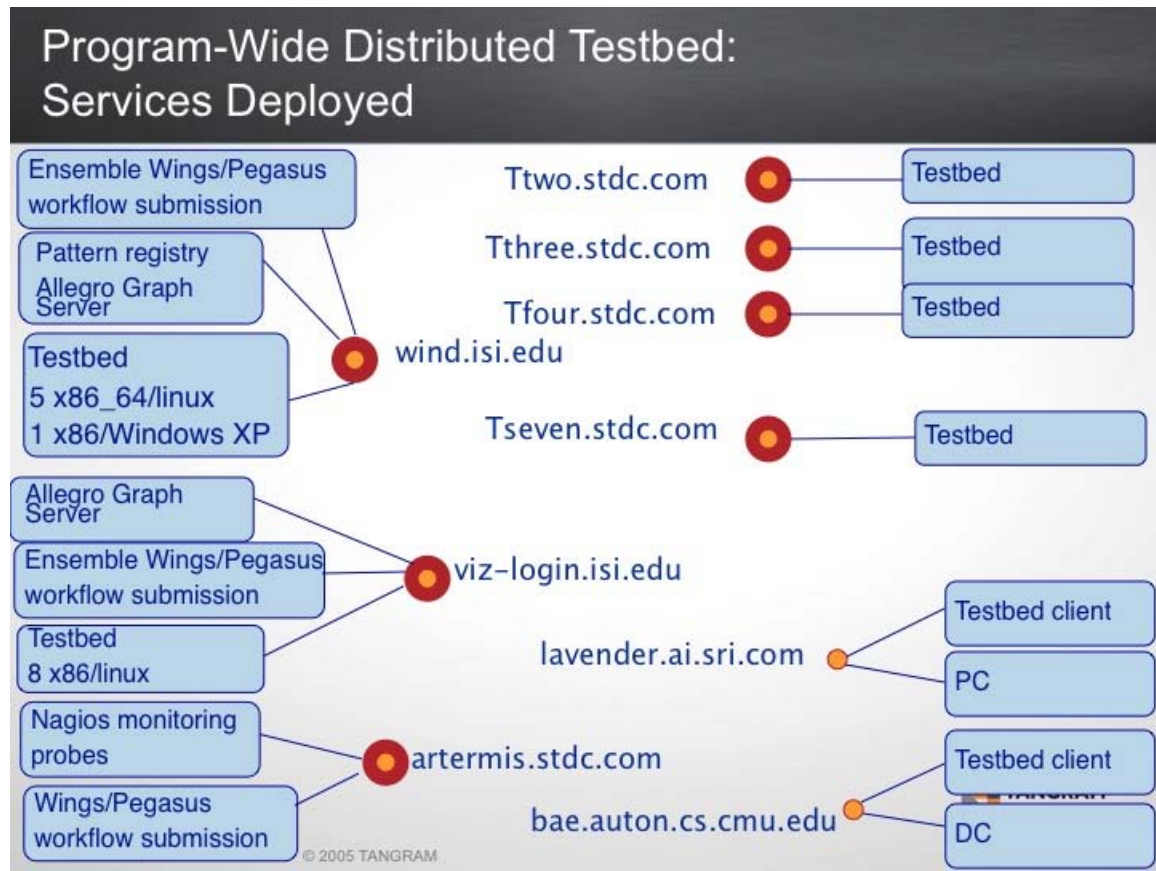
- SE18-TR15: The SE-18 System shall demonstrate the ability to track lineage (audit trail) of assertions and hypotheses during workflow processing. The Workflow System Logging will support necessary audit logging.
- SE18-TR17: The SE-18 System shall demonstrate the ability to orchestrate workflow components of more than one process type. This is supported by the automated workflow generation algorithm.
- SE18-ER4: The SE-18 System shall demonstrate the ability to execute workflow components on multiple operating system types and versions. This is supported by the existing workflow mapping algorithm. SR deployed an ISI testbed with a cluster containing a variety of architectures and operating systems including x86 and x86\_64 architectures with Debian and Redhat Linux OS as well as Windows XP.

## 6. SUPPORT FOR PROGRAM EVALUATIONS

SR has supported program-wide evaluations with testbed setup and workflow system logs. The following sections describe each.

### 6.1 Program Testbed

The SR team has worked on setting up and operating a program-wide testbed. All the program participants are sharing resources in the testbed, including computing and storage resources, software and services, and data sources. Figure 11 shows the Tangram Testbed at the end of the program.



**Figure 11: Tangram Testbed**

The first milestone of the program testbed activity was to include at least one node from each program component and at least one node in the Research and Development Experimental Collaboration (RDEC) facility and to install the workflow system and grid software to test initial connectivity. This was accomplished on January 15, 2007.

Currently two active testbeds operate at ISI in the form of a Viz Cluster, an 8 node x86 Cluster running Debian linux and a Wind Cluster, a 5 node x86\_64 linux cluster running Fedora Core and one x86 Windows XP node.

The testbed is also deployed at SEA on artemis.stdc.com and ttwo, tthree, tfour and tseven.stdc.com nodes.

Changes from SR12 include adding support for Allegro Lisp and Allegro Graph servers on all testbeds and the Ganglia monitoring system running at SEA which monitors all the nodes and services in the Tangram Testbed.

## 6.2 Logging Support

The Tangram program adopted Log4j<sup>1</sup> a widely used logging library for Java, as its underlying logging mechanism. Log4j compatible libraries are available for a number of other programming languages including C, Perl, and Python. Log4j provides a number of features including the ability to dynamically redirect logging statements to different locations, for example, a local file or SEA's logging server as well as filtering out logging messages based on their level of detail. While Log4j provides an infrastructure for integrating logging with applications, it does not specify the format, structure, or content of those logs. Because Tangram has a distributed architecture, it is necessary to be able to correlate logs generated by multiple components within the architecture. To facilitate this activity, SR has defined a logging format and ontology. Additionally, the Workflow Generation API was extended to support the passing of appropriate logging related identifiers to the Data Catalog, Process Catalog and wrapped components. Finally, SR developed libraries that assist in the creation of logs that are compatible with the specified format and logs.

### 6.2.1 Logging Format

We adopted a format for logs from the technical report Grid Logging: Best Practices Guide<sup>2</sup> produced by the Center for Enabling Distributed Petascale Science<sup>3</sup>. This format has a number of benefits including a simple to parse format, extensibility, and scalable processing [Gunter et al., 2005]. Log messages are defined in terms of key=value pairs. In each log message, there are a number of required pairs, which are as follows (Note that the key is given in parenthesis).

- A time stamp (ts) that specifies when the log statement was generated. The timestamp is specified in terms of the ISO8601 time standard [ISO-8601, 1888]. All times are given in UTC.
- A message identifier (msgid) that uniquely identifies each log message.
- An event name (event) that defines the type of event that this log message pertains to. Events can be seen as a program activity. For example, during the planning of a workflow, Pegasus selects the sites at which jobs should be executed, this event or program activity is called site selection and is given the event type `event.pegasus.siteselection`. Thus, all log messages generated by Pegasus during site selection would have the site selection event type.
  - An event name may have suffix, either “.start” or “.end”, appended to it to denote the beginning and end of an event.
  - Each event name is defined in a namespace that begins with event and includes the program or software component that the event occurs in.

---

<sup>1</sup> <http://logging.apache.org/log4j/>

<sup>2</sup> <http://www.cedps.net/images/f/fd/CEDPS-troubleshooting-bestPractices.pdf>

<sup>3</sup> <http://www.cedps.net/>

- An event identifier (eventId) that uniquely identifies this event from other events. This event identifier is shared across log messages for the same event so that messages about the same event can be associated with one another.
- A key=value pair that defines the entity/data that is being processed during the event. For example, during planning, Pegasus is operating on a workflow instance, known as a dax. Thus, each log message would include the pair `dax.id=<identifier for the dax>`.

Beyond these required key=value pairs, log messages can have additional key=value pairs called event attributes that allow additional information about the events to be extracted from log messages.

### 6.2.2 Logging Ontology

The Logging Ontology defines the keys and the types for the values associated with those keys. We now highlight the core parts of the ontology. A full list of ontology terms can be found on the tangram wiki site. The definitions provided in this ontology were motivated by another ontology designed to organize performance data for Grid-based workflow systems [Truong et al , 2007]. In the previous section, we introduced the three core elements of the ontology: entity identifiers, event types, event attributes.

The entity identifiers specified in the Logging Ontology map to the major data structures that are handled by the workflow system. To ensure that these data items can be found within the logs, universally unique ids are used. To ensure uniqueness, we use identifiers that follow the RFC4122 specification<sup>4</sup> but may have additional strings prepended or appended to the identifier.

The following entity identifiers are defined by the Ontology: Request Portfolio Identifier (portfolio.id), Workflow Request Identifier (request.id ), Workflow Instance Identifier (dax.id), Executable Workflow Identifier (dag.id), Job Identifier (dag.id + job.id).<sup>5</sup>

These entity identifiers are key to being able to track the provenance of the output of running a portfolio. If the hierarchy of ids from the Job Identifier to the Portfolio Identifier is maintained, then when analyzing the logs, a user can trace back to the portfolio that initiated the production of a particular result. To ensure that this hierarchy is maintained, we introduce a specialized type of log message called

`event.id.creation`, which enables developers to specify the inheritance relationship between entities. The format of this message is as follows:

```
parent.id.type = (key of the parent id)
parent.id = (value of the id)
child.ids.type = key of the child id
child.ids = {value of the child id, ...}6
```

<sup>4</sup> <http://www.ietf.org/rfc/rfc4122.txt>

<sup>5</sup> To provide context to a job.id, it is always paired with the id of the executable workflow that the job pertains to.

<sup>6</sup> Children are assumed to be of the same type.

The second element of the ontology is the events within the SR system. These events correspond to the major processing activities within the processing and execution of a portfolio. These include events for the management of portfolios (i.e. the ensemble manager), workflow generation, planning, and finally execution.

The final element of the ontology is the various additional event attributes that can be found within log messages. Examples of these attributes, include the hostname and operating system on which the workflow system runs (system.hostname, system.os), the contents of queries to the Process Catalog and Data Catalog (query.input), as well as optional human readable messages (msg). Additionally, SR generates event attributes for information gathered during job execution by the job wrapping mechanism Kickstart [Voeckler et al., 2006].

Bellow is an illustration of a log message following the Logging Format and Ontology we have defined:

```
ts = 2007-12-08T18:39:19.372375z
msgid = 77285E73-49AB-4EAB-AFED-BFCA90E4CEF4
event = event.pegasus.siteselection.start
eventId = 9AA64C69-D449-428A-8FBC-F46C8E237F40
dax.id = 550e8400-e29b-41d4-a716-446655440000
prog = "Pegasus"
system.hostname = prov.isi.edu
msg = "Doing site selection"
```

### **6.2.3 Extension to the Workflow Generation API for Logging**

In order to ensure that logs generated by different components within the Tangram architecture can be successfully correlated, we modified the Workflow Generation API to pass the appropriate entity identifiers (called logging data identifiers within the API) to all external components including the underlying wrapped components. Thus, from the logs, one can see how the processing and execution of a portfolio impacts all the components within the Tangram system.

### **6.2.4 Client Side Logging Library**

To ease the integration of this logging format with the existing Tangram software, we developed a library that supports the generation of properly formatted log messages. Additionally, the library supports the tracking of events, creation of log messages for those events and automatic generation of some identifiers as well as time stamps. Using this library, developers do not have to track event id information or repeat entity identifier information for every log message. While this library was developed for internal use by SR, it has also been adopted by other program participants.

## 7. SOFTWARE RELEASES

ISI has developed and released three core pieces of software that correspond to the architecture shown in Figure 1: Ensemble Manger, Wings (i.e. workflow generation), and Pegasus (i.e. workflow ranking, mapping and execution). Here we provide installation instructions for each of these software components. In addition to these instructions, we provide instructions for installing the Grid Services (such as Condor and Globus) that are required to execute workflows on the TanGrid.

### 7.1 Installation of the SR Workflow System

This section provides installation information for the SR workflow system.

#### 7.1.1. Required Software

These are required for installing the SR workflow system.

- ANT
- JAVA 1.5+
- PEGASUS 2.2.0CVS
- WINGS
- CONDOR 7.1.0 only.
- MYSQL

#### 7.1.2. Build and Install Wings

##### Fetch Wings Code

```
% svn co  
https://tangram.stdc.com/svn/SystemResearch/branches/wings/sel8-rev2
```

##### Wings Compilation, Installation & Running

```
% cd sel8-rev2  
  
% ant jar  
- Creates wings.jar in lib/ directory (and creates a build/ directory)  
  
% ant clean [optional]  
- Removes the build/ directory  
  
% chmod 755 awg  
- Set execute permissions on the "awg" file  
  
% export WINGS_HOME=/path/to/wings/dir [optional]
```

- The main wings directory (directory containing the "awg" file)
- If not set, "awg" script will automatically set WINGS\_HOME to the directory where it is located

```
% export PATH=$WINGS_HOME:$PATH [optional]
```

- If you would like to put the "awg" script in your shell path
- If you didn't set WINGS\_HOME above, then use:
 

```
% export PATH=/path/to/wings/dir:$PATH
```

```
% vi wings.properties (or any other editor) [optional]
```

Check Property "ontology.root.dir"

- Change it's value if you would like to use a local copy of the wings ontology, templates, and seeds.
- If not set (or if the set directory does not exist), it defaults to <wings\_home>/ontology

- The following properties can be set via the properties file as well as overridden via the command line.
  - logs.dir
  - output.dir

```
% awg -h (to check options to the script)
```

Subnote:

- The seed files are present in the ontology/se18/seeds directory
- The template files are present in the ontology/se18 directory

### Seed Validation

```
% awg -v -s <seed name>
```

- This will print out the seeded template in OWL as Wings understands it.
- A User can then look at the interpreted template to make sure it corresponds to what the user wants.

### Example Runs

```
% awg -s SE18-SingleGroupDetector-Tangram
```

- awg simply runs the seed "SE18-SingleGroupDetector-Tangram" with default options and the default configuration file from <wings\_home>/wings.properties

```
% awg -c $HOME/wings.properties -s SE18-SingleGroupDetector-Tangram -l /tmp/logs -o /tmp/output
```

- This means that awg runs the seed "SE18-SingleGroupDetector-Tangram" and stores the logs in /tmp/logs directory, and the output daxes in /tmp/output directory, and it picks up a local configuration file from \$HOME/wings.properties

### Provenance DB Installation (Optional)

The workflow generation provenance data is currently stored in a database at `seagull.isi.edu`.

If a local database is required, then please use the file `"wgpc.sql"` to populate the database. Then, edit the `wings.properties` file and modify `wgpc` database properties.

Note that this provenance data is different from what is sent over to SEA via log files. That will continue to be sent over via `log4j`.

### Set env WINGS\_HOME to the Wings directory

```
$ export WINGS_HOME=<path to wings>
```

- `source $WINGS_HOME/setenv.sh` (if your shell is bash)

```
$source $WINGS_HOME/setenv.sh
```

## **7.1.3. Build Pegasus**

### Download Pegasus from SVN at

```
https://tangram.stdc.com/svn/SystemResearch/branches/pegasus/current/
```

### Set env PEGASUS\_HOME to the Checkout Directory

```
$export PEGASUS_HOME=<path to pegasus-svn-checkout>
```

```
source $PEGASUS_HOME/setup-devel.sh (if your shell is bash)
```

```
$ source $PEGASUS_HOME/setup-devel.sh
```

### Build pegasus using ant

```
$ ant clean dist
```

## **7.1.4. Install Pegasus**

### Copy \$PEGASUS\_HOME/dist/pegasus-\*.tar.gz and untar it

```
$ cp $PEGASUS_HOME/dist/pegasus-binary-*.tar.gz /tmp $ cd <path to software installation directory> $ gtar zxvf /tmp/pegasus-binary-*.tar.gz
```

Set env PEGASUS\_HOME to the binary installation path

```
$ export PEGASUS_HOME=</path to binary install>
```

Remove wings jar included in Pegasus

```
$ rm $PEGASUS_HOME/lib/wings.jar
```

### **7.1.5. Build Ensemble Manager**

Download Ensemble Manager code from SVN at

<https://tangram.stdc.com/svn/SystemResearch/branches/ensemble/current/>

Set env ENSEMBLE\_HOME to the checked out directory

```
$ export ENSEMBLE_HOME=<path checked out>
```

Source \$ENSEMBLE\_HOME/setup-devel.csh if your shell is CSH or setup-devel.sh if your shell is BASH

```
$ source $ENSEMBLE_HOME/setup-devel.sh
```

Run ant clean package

```
$ant clean package
Buildfile: build.xml
```

```
clean:
```

```
  [delete] /nfs/asd2/gmehta/jbproject/Ensemble/dist not found.
  [delete] /nfs/asd2/gmehta/jbproject/Ensemble/build not found.
```

```
init:
```

```
  [mkdir] Created dir:
/nfs/asd2/gmehta/jbproject/Ensemble/dist/ensemble
  [mkdir] Created dir: /nfs/asd2/gmehta/jbproject/Ensemble/build/src
  [echo] full ISO timestamp:
```

```
compile:
```

```
  [javac] Compiling 28 source files to
/nfs/asd2/gmehta/jbproject/Ensemble/build/src
```

```
...
```

```
...
```

```
  [mkdir] Created dir:
/nfs/asd2/gmehta/jbproject/Ensemble/dist/ensemble/var
  [copy] Copying 5 files to
/nfs/asd2/gmehta/jbproject/Ensemble/dist/ensemble
```

```
[gzip] Building:
/nfs/asd2/gmehta/jbproject/Ensemble/dist/ensemble.tar.gz
[delete] Deleting:
/nfs/asd2/gmehta/jbproject/Ensemble/dist/ensemble.tar
```

A tarball will be created in \$ENSEMBLE\_HOME/dist/ensemble.tar.gz

### 7.1.6. Install Ensemble Manager

Copy the binary tarball built in the earlier step to an installation location and untar it

```
$ gtar zxvf $ENSEMBLE_HOME/dist/ensemble.tar.gz
```

Set environment ENSEMBLE\_HOME to the untarred directory

```
export ENSEMBLE_HOME=<path to ensemble binary directory>
```

Configure other paths in correct order

```
$ unset CLASSPATH
```

```
$ source $WINGS_HOME/setenv.sh
```

```
$ source $PEGASUS_HOME/setup.sh
```

```
$ source $ENSEMBLE_HOME/setup.sh
```

### 7.1.7. Create the Ensemble Data Base (DB)

As user root create a db for storing the ensemble schema in MySQL

```
create database <databasename>;
```

Add a username and password which has access to this db

```
grant all on <databasename>.* to <username>@"<hostname>" identified by
"<password>";
```

```
flush privileges;
```

Populate the created Db with the ensemble schema from  
\$ENSEMBLE\_HOME/sql/ensemble.sql

```
mysql -u <username> -p databasename < $ENSEMBLE_HOME/sql/ensemble.sql;
```

### 7.1.8. Edit the Ensemble configuration file

Edit the \$ENSEMBLE\_HOME/etc/properties or create a file \$HOME/.ensemblerc

```
condor.home=<path to condor install home directory>
pegasus.home=<path to pegasus install home directory $PEGASUS_HOME>
wings.home=<path to wings install>
ensemble.db.url=<jdbc url to ensemble db . e.g.
jdbc:mysql://smarty.isi.edu/ensembledb>
ensemble.db=MySQL
ensemble.db.user=<dbusername>
ensemble.db.password=<dbpassword>
ensemble.localdir=<path where the ensembles workflows are planned and
dags are generated. Default is $ENSEMBLE_HOME/var>
```

### 7.1.9. Edit Log4j.configuration file

```
log4j.rootCategory=DEBUG, File, Console, Socket
```

```
log4j.logger.anchor.datametrics=OFF
log4j.logger.com.hp.hpl.jena=OFF
log4j.logger.org.griphyn=DEBUG
log4j.logger.pegasus=DEBUG
log4j.logger.edu.isi=DEBUG
```

```
#
```

```
# The default file appender
```

```
#
```

```
log4j.appender.File=org.apache.log4j.RollingFileAppender
log4j.appender.File.Threshold=DEBUG
log4j.appender.File.File=/tmp/ensemblemanager.log
log4j.appender.File.layout=org.apache.log4j.SimpleLayout
log4j.appender.File.Append=true
log4j.appender.File.MaxFileSize=100MB
```

```
#
```

```
# Console Appender
```

```
#
```

```
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.SimpleLayout
log4j.appender.Console.Threshold=INFO
```

```
#
```

```
# The Socket Appender
#
log4j.appender.Socket=org.apache.log4j.net.SocketAppender
log4j.appender.Socket.Threshold=INFO
log4j.appender.Socket.RemoteHost=artemis.stdc.com
log4j.appender.Socket.Port=40940
log4j.appender.Socket.ReconnectionDelay=5000
log4j.appender.Socket.LocationInfo=true
```

The sample log4j.properties shown here is shipped in \$ENSEMBLE\_HOME and is used by default. You can modify any of the properties in the \$ENSEMBLE\_HOME/log4j.properties or provide an alternative log4j.properties file by passing the option -Dlog4j.configuration=file:/path/to/log4j.properties/file

Note that you need to add the following entries to any non standard log4j.properties file for all the logging to appear correctly.

```
log4j.logger.anchor.datametrics=OFF
log4j.logger.com.hp.hpl.jena=OFF
log4j.logger.org.griphyn=DEBUG
log4j.logger.pegasus=DEBUG
log4j.logger.edu.isi=DEBUG
```

## 7.2 Grid Services

This section describes how to set up a node in the program testbed, get certificates, and install the workflow system and grid software.

### 7.2.1 Linux Headnode Installation

#### Required Software

Head Node/Server (Each site should have at least one head node and several worker nodes)

OS : Linux native or Linux VM.

Software packages :

1. Ant 1.7
2. Java 1.6
3. Allegro Lisp, Allegro Cache, Allegro Graph 8.1 with latest updates
4. ganglia 3.0.6 or 3.0.7

5. condor 7.1.3
6. globus 4.0.7 or 4.0.8
7. Tangram Software stack

The easiest way to install for several machines if they share a shared file system is to install the software on the shared file system.

e.g. /nfs/software

create directories for each package eg.

cd /nfs/software

mkdir -p ant/src java/src ganglia/src condor/src globus/src

### Install Ant

Download ant

cd /nfs/software/ant/src wget <http://wind.isi.edu/software/apache-ant-1.7.0-bin.tar.gz>

cd /nfs/software/ant tar zxvf src/apache-ant-1.7.0-bin.tar.gz

symlink the installation directory to default

ln -s apache-ant-1.7.0 default

add the following paths to /etc/profile

export ANT\_HOME=/nfs/software/ant/default export  
PATH=\$ANT\_HOME/bin:\$PATH

### Install Java

Download java

cd /nfs/software/java/src

wget [http://wind.isi.edu/software/jdk-1\\_5\\_0\\_15-linux-i586.bin](http://wind.isi.edu/software/jdk-1_5_0_15-linux-i586.bin)

wget <http://wind.isi.edu/software/jdk-6u5-linux-i586.bin>

chmod 755 \*.bin

cd /nfs/software/java

sh src/jdk-1\_5\_0\_15-linux-i586.bin

Read the License. Type Yes at the end. The automated installer will install java 1.5

Symlink the jdk 1.5 to default

sh src/jdk-6u5-linux-i586.bin

Read the License. Type Yes at the end. The automated installer will install java 1.6

## Install Ganglia

Ganglia is a monitoring system that needs to run on each node. It consists of 3 parts:

1) gmond daemon that needs to be installed on each node, 2) gmetad that needs to be run on the head node or one of the nodes as a gatherer and 3) ganglia-web module which displays the information and graphs.

Ganglia is available for most distributions via standard yum and apt repositories. If you can't find them you can install tarballs from <http://www.ganglia.org>. A system admin just needs to run the command on all nodes. Depending on your Linux installation you may need the following extra packages perl-Compress-zlib perl-XML perl-XMLParser Sqlite

```
yum install gmond
```

or

```
apt-get install ganglia-monitor libganglia1
```

And then on the head node:

```
yum install gmetad
```

or

```
apt-get install gmetad
```

Also:

```
yum install ganglia-web
```

or

```
apt-get install ganglia-web
```

If you cannot find these packages you may want to do

```
yum list ganglia
```

or

```
apt-cache search gmond gmetad ganglia
```

You can also download the ganglia packages from the ganglia website and install them manually.

After installation copy the gmond.conf file from <http://wind.isi.edu/software/gmond.conf> to /etc/gmond.conf on each node

- Modify the gmond.conf file for your setup.
  - Edit the following section and put values for your cluster.

```
cluster {

    name = "Windward"
    owner = "ISI / CGT"
    latlong = "N30.0 W122.23"
    url = "http://wind.isi.edu/ganglia"

}
```

- edit the `udp_send_channel` section and change the `mcast_join` hostname/ip to be the hostname where your gmetad daemon is running.

```
udp_send_channel {

    mcast_join = wind.isi.edu
    port = 8649
    ttl = 1

}
```

- Copy the gmetad.conf file from <http://wind.isi.edu/software/gmetad.conf> to `/etc/gmetad.conf` on the node running the gmetad daemon.
  - change the gridname to your Grid name.
  - Change `trusted_hosts` to add `wind.isi.edu` and `128.9.72.178` if not already there.
  - Send to `gmehta@isi.edu` the host/ip info where your gmetad is running.
- Start all the gmond daemons by running the `/etc/init.d/gmond` and `/etc/init.d/gmetad` scripts.
- If you installed the ganglia-web package then you may additionally need to start your httpd server to show graphs. Otherwise your site will still be displayed on the `wind.isi.edu/ganglia` url

### Install Condor

- Download and untar condor version 7.1.3 from <http://cs.wisc.edu/condor> for your linux os and architecture. Download the dynamic tar.gz packages instead of RPM's
- You may additionally need to install `compat-libstdc++` libraries if your condor installation does not work. You will get an error when you try to start condor and this will be in the logs.
- After untaring in `/nfs/software/condor/src/condor-7.1.3` run the `./condor-configure` script
  - `./condor_install --install-dir <path to installation> --make-personal-condor --owner=condor`

(the make personal condor is only if you are running a single node cluster.

*If you plan to install condor for multiple nodes, then you may want to make the head node just act as a submit node --type=submit and select some other node to be a central manager which normally does not run any job. All the cluster nodes will then be of type=execute.)*

- The script may prompt you to specify local condor directory. Set them to a non shared file system.
- If you plan to install condor also as a scheduler for various nodes in the cluster you will need additional configuration on each node.
- On each node set the CONDOR\_HOST to a machine acting as the central manager. This is generally the machine where you ran the condor-configure command as --make-personal-condor or as --type=manager.

Check the condor\_config file written in path-to-installation/etc/condor\_config

You may additionally need to change an entry in the condor\_config file where it says HOSTALLOW\_WRITE to be \*.yourdomain, \*.isi.edu, \*.stdc.com

```
Set env variables PATH=<path to condor/bin>:<path to condor/sbin>:$PATH
CONDOR_CONFIG=<path to condor/etc/condor_config>
```

Once condor is installed run the the command as root.

```
/nfs/software/condor/<path where condor was installed>/sbin/condor_master
```

You should see several condor daemons start up. e.g. master, collector, negotiator( if your machine is set to manager), schedd (if your machine is set to submit), startd (if your machine is set to execute) or all of them if you chose (personal-condor).

You can either write a inetd script to start your condor automatically at start time or modify the file in /nfs/software/condor/<install>/examples/condor.boot and install it in the appropriate inetd locations.

### Install Globus

Download globus 4.07 binary from [www.globus.org](http://www.globus.org) for your system.

save it in /nfs/software/globus/src

- Untar the binary tarball
- Run the command
  - ./configure --prefix=/nfs/software/globus/4.0.7 --enable-wsgram-condor --disable-tests --disable-wstests

(If you are running torque/pbs or some other scheduler other than condor you will need to do --enable-wsgram-pbs or --enable-wsgram-lsf etc.)

- Globus will be installed in /nfs/software/globus/4.0.7.
- Make a symlink from 4.0.7 to default
- If you install globus on a shared file system you may want to move the globus/var and globus/tmp directories to a local file system and symlink them from the installation directory e.g.

```
cd /nfs/software/globus/4.0.7 mkdir /var/spool/globus mv var
/var/spool/globus/var mv tmp /var/spool/globus/tmp
```

```
ln -s /var/spool/globus/var var ln -s /var/spool/globus/tmp tmp
```

- You will need to write several files to enable globus services.
- To start with
  - Write a file called globus-gatekeeper in /etc/xinetd.d directory

```
service globus-gatekeeper
{
    socket_type    = stream
    protocol      = tcp
    wait          = no
    user          = root
    server         = /nfs/software/globus/default/sbin/globus-
gatekeeper
    server_args    = -conf /nfs/software/globus/default/etc/globus-
gatekeeper.conf
    disable       = no
    env           = LD_LIBRARY_PATH=/nfs/software/globus/default/lib
    env           += GLOBUS_LOCATION=/nfs/software/globus/default
    env           += GLOBUS_TCP_PORT_RANGE=40000,41000
}
```

- Write a file called gridftp in the same directory

```
service gridftp
{
    instances      = 100
    socket_type    = stream
    wait          = no
    user          = root
    server         =
/nfs/software/globus/default/sbin/globus-gridftp-server
    server_args    = -i -d info -l
/var/spool/globus/var/gridftp.log
    log_on_success += DURATION USERID
    log_on_failure += USERID
    nice          = 10
    disable       = no
}
```

```

        env                +=
GLOBUS_LOCATION=/nfs/software/globus/default
        env                +=
PATH=/nfs/software/globus/default/bin:/nfs/software/globus/default/sbin
        env                +=
LD_LIBRARY_PATH=/nfs/software/globus/default/lib
        env                += GLOBUS_TCP_PORT_RANGE=40000,41000
    }

```

- edit all the paths to the globus software mentioned in the above file for your environment.
- edit the GLOBUS\_TCP\_PORT\_RANGE to define the ports which you have poked in your firewall.
- edit the file /etc/services and add the lines

gridftp 2811/tcp globus-gatekeeper 2119/tcp

- Restart xinetd
- Start the gsissh server by first copying the file  
/nfs/software/globus/4.0.7/sbin/SXXsshd to /etc/init.d/gsisshd
- cd /etc/init.d
- Run /sbin/chkconfig --add gsisshd
- Edit /nfs/software/globus/4.0.7/etc/ssh/sshd.conf
- Uncomment the port line on the top and change it from 22 to 40022
- Start the gsissh server by running the script as root /etc/init.d/gsisshd start

### */etc/grid-security*

Download the package <http://wind.isi.edu/software/grid-security.tar.gz>

Untar the package as root in /etc.

This will create directory called grid-security with the CA certificates etc in place.

### ***GRIDMAP file***

A file named grid-mapfile has to be created to map DN credentials to local users on the node.

The file format is

"/DN/FOO/BAR" userid "/DN/BAR/FOO" userid2

The allocated user DN's are mentioned below. This will be later provided as an auto update file which every site can download using wget in a cron job.

## ***TESTING GLOBUS***

Make sure you set your environment variables to include

GLOBUS\_LOCATION=</path to globus dir>

and source \$GLOBUS\_LOCATION/etc/globus-user-env.sh

After you have installed your User and host certs as described below, you need to run the command

grid-proxy-init

This will generate a proxy valid for 12 hours

Then Follow the testing the ISI grid instructions at the bottom.

To test your own grid server just change the hostname to your hostname.

### **7.2.2 Linux Cluster Node Installation**

Follow similar instructions at:

[https://wiki.boozallenet.com/tangram/index.php/SR-SE-18-Linux\\_Cluster\\_node-Instructions](https://wiki.boozallenet.com/tangram/index.php/SR-SE-18-Linux_Cluster_node-Instructions)

### **7.2.3 Windows Node Installation**

Follow similar instructions at:

[https://wiki.boozallenet.com/tangram/index.php/SR-SE-18-Windows\\_node-Instructions](https://wiki.boozallenet.com/tangram/index.php/SR-SE-18-Windows_node-Instructions)

## **8. INTERIM PROJECT REPORTS AND DOCUMENTATION RELEASED**

The following reports and documentation were made available in the program wiki. The software released is described separately in the next section.

- Instructions for modifying and maintaining site Catalogs -- Released Oct 4, 2008.

- Instructions for the Ensemble Manager installation -- Released September 15, 2008.
- Instructions for template validation in Wings -- Released September 12, 2008.
- SR SE-18 Acceptance Test Cases -- Released July 31, 2008.
- Workflow Generation API Version 2.1 -- Released: June 10, 2008.
- Documentation on workflow requests to task the workflow system -- Released May 16, 2008.
  - Formulating workflow requests for SE-18
  - General documentation and background on workflow requests
- Workflow Generation API for SE-18 -- Released April 8, 2008.
- SR Critical Design Review for SE18 -- Released March 14, 2008.
- Formalization and algorithm for automated generation of computational workflows from templates -- Released March 14, 2008.
- SR Preliminary Design Review for SE18 -- Released February 22, 2008.
- Design document for a user interface to the workflow system -- Released November 26, 2007.
- Description of SR-12 workflow ranking algorithm -- Released November 9, 2007.
- SR planning materials prepared for the Working Group Session held in Chicago IL on November 8-9, 2007.
- SR\_Workflow\_Generation\_API\_V1.9 - Released September 4, 2007.
- Workflow System Logs description - Released July 20, 2007.
- SR planning materials prepared for the Working Group Session held in Chicago IL on July 19, 2007:
  - SR plans for SR-12, SE-18, and beyond
  - SR high-level architecture
  - SR thoughts on other program requirements, architecture and design issues
- Ensemble Manager description -- Released July 13, 2007
- SR\_Workflow\_Generation\_API\_V1.85 -- Released July 11, 2007.
- SR\_Workflow\_Generation\_API\_V1.8 -- Released June 3, 2007.
- Basic Component Encapsulation -- Basic information to describe the encapsulation of executable codes and associated wrappers as workflow components. Released April 13, 2007.
- SR\_Workflow\_Generation\_API\_V1.1 -- Released April 5, 2007.
- SR6\_Metadata\_Notes -- Metadata analysis for SR-6 group finding workflows. Released March 30, 2007.

- Running SR-6 workflows -- Detailed instructions to create and run SR-6 workflows. Released March 29, 2007.
- Wings Workflow Creation System -- Instructions to install the Wings software, which includes a workflow template browser/editor and an automatic workflow planning and generation capability that creates workflow instances to submit to Pegasus. Released March 22, 2007.
- SR-6 Demonstration Report - reviewed at the SR March 1, 2007 site visit.
- Workflow Generation API - Describes the current draft of the proposed API for SR's automatic workflow generation capability. This API does not cover other reasoners needed by SR for workflow template editing, workflow template validation, workflow visualization, and other aspects of workflow management.
- Automated Workflow Generation Process - Describes the approach to SR's automatic workflow generation capability.
- Wings/Pegasus Overview - Describes algorithm and data models used in the current implementation of the Wings/Pegasus workflow system.
- 20061214-SR-GroupSubWorkflowVariants-v2.pdf -- a report describing the workflows demonstrated in December 2006 by SR.
- 20061128-SR-SR6-DataConops-v1.pdf -- describes the conops for the data sources used in the workflow. It also outlines the assumptions made regarding the data.
- 20061102-SR-HatsBatchData-v1.pdf -- a description of the data generated by the Hats simulator.
- Link to the Hats simulator web site used for synthetic data generation in SR-6.
- conops-SR6.doc -- a draft overall conops for the six-month demo from October 31, 2006. It also describes the design of the workflow and the algorithms and data to be used. The overall goals for SR-6 evolved in later months so those portions of this document are dated.
- 061213-SR-AlgorithmInfoRequiredForGrid-v3.doc -- Describes what information is needed by Pegasus from the Algorithm Catalog.
- Best Practices for Writing Codes of Workflow Components -- Describes best practices for developing application codes to be run on the Grid. This was originally described in the document 061201-SR-RunningCodesGrid-v2.doc.
- Componentizing Legacy Codes -- Examples of how to wrap legacy codes for execution on the Grid. It reports lessons learned from the process of componentizing third party codes for the SR-6 demo, and includes a section on building standalone executables in Lisp. This was originally described in the document 061208-SR-ComponentizingCodes-v4.pdf.
- FAQ on setting up grid nodes -- Extensive documentation on program testbed, including how to set up a grid node, how to obtain certificates, how to install grid services and workflow system.

- Globus web site (<http://www.globus/>) includes many accessible materials, papers, and software downloads.
- Ontologies and catalogs used in Wings/Pegasus (<http://vtcpc.isi.edu/provenance>) shown for an example workflow, including pointers to OWL domain ontologies, OWL component models, metadata, and representations for a workflow template, a workflow instance, a DAX, and an executable workflow.
- Pegasus web site (<http://pegasus.isi.edu/>) -- includes many descriptive papers and software download.
- Wings web site (<http://www.isi.edu/ikcap/wings/>) -- includes papers and detailed information about the approach.

## 9. CONCLUSIONS

This Program was divided into Phases. In the first phase the Program concentrated on building the Core Functionality, which included the basic grid infrastructure and basic workflow composition and execution functionality. The second phase was focused on developing the Feasibility functionality. This included plug and play analytic components, data characterization services, component process characterization and greatly improved core functionality. The third phase was to be focused on Core Intelligence, emphasizing the development and testing of complete computational analytic workflows that serve as threat detectors and alerting functions. Due to funding cutbacks the Program did not continue to the third phase. We did however, achieve significant success and did prove the feasibility of the workflow concept.

## 10. REFERENCES

1. Ashley, K. D. and Aleven, V, 1997, Reasoning symbolically about partially matched cases. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence. San Francisco, CA: Morgan Kaufmann, pp. 335–341.
2. Baader, F. and P. Narendran, "Unification of Concept Terms in Description Logics", Journal of Symbolic Computation, 2001.
3. Baader, F. C. Lutz, M. Milicic, U. Sattler, and F. Wolter. "Integrating Description Logics and Action Formalisms: First Results", Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05), Pittsburgh, PA, USA, 2005.
4. Bergmann R and Stahl, A, 1998, Similarity measures for object-oriented case representations. In Proceedings of the Fourth European Workshop on Case-Based Reasoning. Berlin: Springer, pp. 25–36.
5. Champin, PA and Solnon, C, 2003, Measuring the similarity of labeled graphs. In Proceedings of the Fifth International Conference on Case-Based Reasoning. Berlin: Springer, pp. 80–95.
6. Condor Dagman. <http://www.cs.wisc.edu/condor/dagman>, 2008.
7. Gunter, D.K. and Tierney, B. L. Scalable Analysis of Distributed Workflow Traces, The 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05)
8. Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Blackburn, K., Lazzarini, A., Arbree, A., Cavanaugh, R., and Koranda, S. "Mapping Abstract Workflows onto Grid Environments." Journal of Grid Computing, Vol. 1, No. 1, 2003.
9. Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M., Vahi, K., Livny, M., Pegasus: Mapping Scientific Workflows onto the Grid. In Across Grids Conference 2004.
10. Deelman, E., Singh, G., Su, M., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C., and D. S. Katz. "Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems". Scientific Programming Journal, Vol 13(3), 2005.
11. Deelman, E., and Gil, Y. (Eds). "Final Report of the NSF Workshop on Challenges of Scientific Workflows", National Science Foundation, Arlington, VA, May 1-2, 2006. <http://www.isi.edu/nsf-workflows06>.
12. Forbus, K, Gentner, D and Law, K, 1994, MAC/FAC: a model of similarity-based retrieval. Cognitive Science 19(2), 141–205.
13. Gil, Y., Ratnakar, V., and Deelman, E. Virtual Metadata Catalogs: Augmenting Metadata Catalogs with Semantic Representations. Fourth International Semantic Web Conference (ISWC-05), Galway, Ireland, November 7-10, 2005.
14. Gil, Y. "Workflow Composition". In Workflows for e-Science, D. Gannon, E. Deelman, M. Shields, I. Taylor (Eds), Springer Verlag, 2006.
15. Gil, Y., Ratnakar, V., Deelman, E., Mehta, G. and J. Kim. "Wings for Pegasus: Creating Large-Scale Scientific Applications Using Semantic Representations of Computational Workflows." Proceedings of the 19th Annual Conference on Innovative Applications of Artificial Intelligence (IAAI), Vancouver, British Columbia, Canada, July 22-26, 2007.

16. Gil, Yolanda, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers. "Examining the Challenges of Scientific Workflows," IEEE Computer, vol. 40, no. 12, pp. 24-32, December, 2007.
17. Goderis, A., Li, P., Goble, C.A.: Workflow discovery: the problem, a case study from e-science and a graph-based solution. In: ICWS, IEEE Computer Society (2006) 312–319.
18. Hull, D., Zolin, E., Bovykin, A., Horrocks, I., Sattler, U., and Stevens, R. "Deciding Semantic Matching of Stateless Services." Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI), 2006.
19. ISO-8601, "Data Elements and Interchange Formats - Information Exchange - Representation of Dates and Times", International Organization for Standardization, 1888 <http://www.iso.ch/markete/8601.pdf>
20. Haluk Topcuoglu, Salim Hariri, Min-You Wu, "Task Scheduling Algorithms for Heterogeneous Processors," hew , p. 3, 1999.
21. Jena, Semantic Web Framework for Java. <http://jena.sourceforge.net>, 2008.
22. Voekler, J. Mehta, G., Zhao, Y., Deelman, E., Wilde, M. .Kickstarting Remote Applications, Presented at GCE06 Second International Workshop on Grid Computing Environments.
23. Kim, J., Deelman, E., Gil, Y., Mehta, G., Ratnakar, V. Provenance Trails in the Wings/Pegasus Workflow System", In Concurrency and Computation: Practice and Experience, Special Issue on the First Provenance Challenge, 2007.
24. Kim, J., Gil, Y., Ratnakar, V. Semantic Metadata Generation for Large ScientificWorkflows. In Proceedings of the International Semantic Web Conference 2006.
25. Li, L., and Horrocks, I., "A software framework for matchmaking based on semantic web technology", Proceedings of the World Wide Web Conference (WWW), 2003.
26. Liskov, B. and Guttag, J. "Abstraction and Specification in Program Development." MIT Press, 1986.
27. Moody 2006. Hats Simulator Batch Data Sets, <https://wiki.boozallenet.com/tangram-/images/f/fb/20061102-SR-HatsBatchData-v1.pdf>, Nov 2, 2006
28. OWL. <http://www.w3.org/TR/owl-guide>, 2008.
29. Pegasus. <http://pegasus.isi.edu>, 2008.
30. Singh, G., Bharathi, S., Chervenak, A., Deelman, E., Kesselman, C., Manohar, M., Patil, S., Pearlman, L. A Metadata Catalog Service for Data Intensive Applications. SC 2003, 2003.
31. Topcuoglu, T., Hariri, S., Wu, M. Task Scheduling Algorithms for Heterogeneous Processors. Heterogeneous Computing Workshop 1999: 3-14.
32. WingsPegasus 2007, Wings/Pegasus for the First Provenance Challenge. <http://vtcp.csi.edu/provenance>, 2007.
33. Wings: Workflow Instance Generation and Selection. <http://www.isi.edu/ikcap/wings>, 2008.
34. WingsPegasus 2008, Wings Pegasus Overview, [https://wiki.boozallenet.com/tangram/index.php/SR\\_WINGS\\_PEGASUS\\_OVERVIEW](https://wiki.boozallenet.com/tangram/index.php/SR_WINGS_PEGASUS_OVERVIEW).
35. SR SR-12 Report wiki, [https://wiki.boozallenet.com/tangram/index.php/SR-SR-12\\_Report](https://wiki.boozallenet.com/tangram/index.php/SR-SR-12_Report)

36. SR tangram wiki, [https://wiki.boozallenet.com/tangram/index.php/System\\_Research](https://wiki.boozallenet.com/tangram/index.php/System_Research)
37. Truong, H., Dustdar, S., Fahringer, T., Performance Metrics and Ontology for Grid Workflows, Future Generation of Computer Systems Elsevier, 2007.

## 11. ACRONYMS

|       |  |
|-------|--|
| API   | Application Programming Interface                                  |
| AC    | Algorithm Catalog  |
| DAG   | Directed Acyclic Graph   |
| DAX   | <b>DAG XML</b> (Directed Acyclic Graph Extensible Markup Language) |
| DC    | Data Catalog   |
| DOD   | Data Object Description  |
| EM    | Ensemble Manager   |
| FR    | Functional Requirements  |
| ISI   | Information Sciences Institute                                     |
| MySQL | My Structured Query Language                                       |
| PC    | Process Catalog  |
| RDEC  | Research and Development Experimental Collaboration Facility       |
| SEA   | System Evaluation Architecture                                     |
| SE-18 | System Evaluation at 18 months                                     |
| SR    | System Research  |
| SR-12 | System Research demonstration at 12 months                         |
| TR    | Technical Requirement  |
| XML   | Extensible Markup Language   |